Safinaz Volpe
Francesco P. Volpe

AVR-Mikrocontroller-Praxis

Befehlssatz, Tools und Anwendungen

Bei der AVR-Familie von Atmel handelt es sich um 8-Bit Mikrocontroller mit RISC-Architektur für die unterschiedlichsten Anwendungen. Der Programmcode wird in einem FLASH-Speicher abgelegt, so daß auch eine In-Circuit-Programmierung möglich ist.

Dieses Buch beschreibt die Architektur und die Peripherie der AVR-Mikrocontroller. Ferner wird der komplette Befehlssatz übersichtlich dargestellt und anschaulich erklärt. Neben Softwaretools, wie Assembler und Simulator, werden auch die für den Entwurf notwendigen Hardwaretools, wie Programmiergerät und Emulator, vorgestellt. Anschließend wird eine AVR-Experimentierplatine besprochen. Das Programmiergerät und die AVR-Experimentierplatine sind als Bausatz erhältlich, so daß der Leser einen schnellen Einstieg in die Programmierung der AVR-Mikrocontroller erhält. Abgeschlossen werden die Betrachtungen mit fertigen Software-Routinen, die z. B. die Ansteuerung von seriellen EEPROMs und einer LCD-Anzeige erlauben.

Das Buch gliedert sich in:
- Aufbau der AVR-Mikrocontroller-Familie und Befehlssatz
- Softwaretools (Assembler und Simulator)
- Hardwaretools (Programmiergerät und Emulator)
- Anwendungen (I2C-Bus, LCD-Anzeige, RS232, CRC-8 u.v.m)

Dem Buch liegt eine CD mit Assembler, Simulator und den Datenblättern von Atmel, sowie dem Sourcecode der erläuterten Anwendungsbeispielen bei.

M. Eng. Sc., Dipl.-Ing. Safinaz Volpe studierte an den Universitäten Kassel und Melbourne Elektrotechnik. Anschließend war sie eine Zeit lang als Redakteurin bei einer Elektronikzeitschrift tätig.

Prof. Dr.-Ing. Francesco P. Volpe studierte an der Ruhr-Universität Bochum Elektronik. Anschließend promovierte er an der Universität Kassel. Derzeit ist er Professor für Mikrocomputertechnik an der Fachhochschule in Aschaffenburg.

Elektor-Verlag, Aachen
ISBN 3-89576-063-3

ELEKTOR

mit CD-ROM

AVR-Mikrocontroller-Praxis

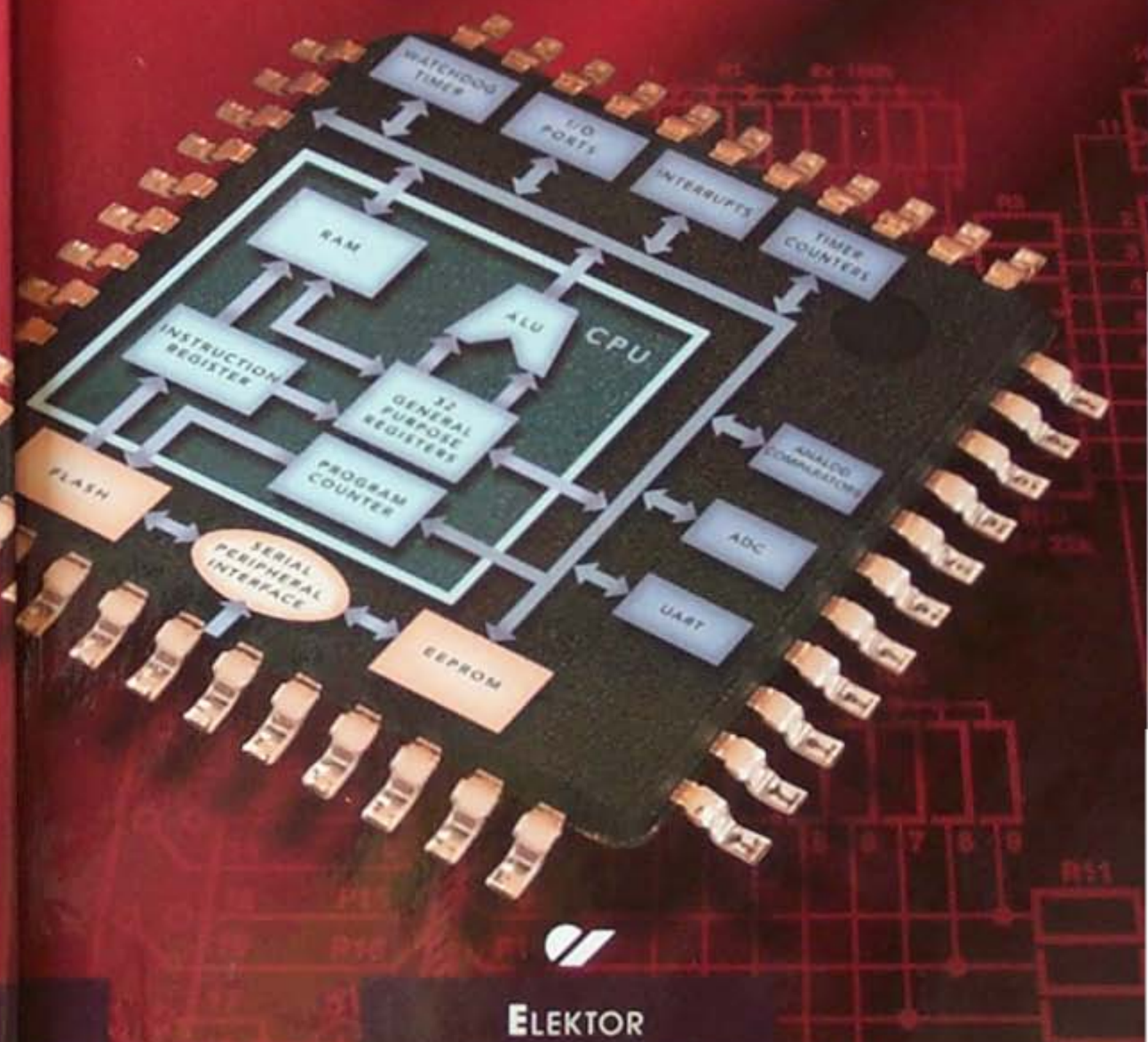
Safinaz Volpe
Francesco P. Volpe

ELEKTOR

Safinaz Volpe
Francesco P. Volpe

AVR-Mikrocontroller-Praxis

Befehlssatz, Tools und Anwendungen



ELEKTOR

Safinaz Volpe
Francesco P. Volpe

AVR-Mikrocontroller-Praxis

Befehlssatz, Tools und Anwendungen

2. Auflage

Elektor-Verlag

Umschlaggestaltung: Ton Gulikers, Segment, Beek (NL)
Grapische Gestaltung: Laurent Martin, Hans Koerfer-Bernstein
Satz und Aufmachung: TechKnow, Hans Koerfer-Bernstein, Aachen
Belichtung: typeline, Aachen und TechKnow, Hans Koerfer-Bernstein, Aachen
Druck: Giethoorn/TenBrink, Meppel, Niederlande

Printed in the Netherlands

ISBN 3-89576-063-3
Elektor-Verlag, 52072 Aachen
989009

Der Elektor-Verlag ist Mitglied der
Verlagsgruppe Segmon B.V. (Niederlande)

Wir widmen dieses Buch unserer Tochter
Sarah Safinaz

Inhaltsverzeichnis

1 Einleitung 7

2 Die AVR-Mikrocontroller-Familie 9

2.1 Eigenschaften und Typen 9
2.2 AVR-Architektur 12
2.3 Speicher und Register 14
2.3.1 Programmspeicher 14
2.3.2 Datenspeicher (SRAM) 16
2.3.3 EEPROM 17
2.3.4 General Purpose Register 18
2.3.5 I/O-Register 20
2.3.6 STATUS-Register 22
2.4 Peripherie 23
2.4.1 I/O-Ports 23
2.4.2 Synchrone serielle Schnittstelle (SPI) 24
2.4.3 Asynchrone serielle Schnittstelle (UART) 26
2.4.4 Analog-Komparator 29
2.4.5 Analog/Digital-Wandler 30
2.4.6 Timer und Counter 32
2.4.7 Watchdog-Timer 38
2.5 Stack 38
2.5.1 Hardware-Stack 40
2.5.2 Stack im Datenspeicher 40
2.6 Reset 41
2.7 Reset und Interrupt-Vektoren 42
2.8 Taktoszillator 43
2.8.1 Quarz-Oszillator 43
2.8.2 Externer Takt 43
2.8.3 Interner RC-Oszillator 44
2.9 Der Befehlssatz 45

3 Software-Entwicklungstools 167

3.1 AVR-Assembler 167
3.1.1 Direktiven 169
3.1.2 Kommentare 170
3.1.3 Ausdrücke 171
3.1.4 Symbole und Marken 172
3.1.5 Datentypen 174
3.1.6 Funktionen 175
3.2 AVR-Simulator 175
3.3 AVR-Studio 176

4 Hardware-Entwicklungstools 181

4.1 AVR-Programmer 181
4.1.1 Hardware des AVR-Programmers 181
4.1.2 Software des AVR-Programmers 187
4.2 Der Emulator AVR AT90ICEPRO 192
4.3 Logikanalysator HP54645D 194

5 Anwendungen 195

5.1 Touch-Memories 199
5.2 Ansteuerung von EEPROMs mit I2C-Bus-Protokoll 215
5.3 Ansteuerung einer LCD-Anzeige 229
5.4 Telefonkartenleser 240
5.5 Magnetkartenleser 248
5.6 Serielle Kommunikation 261
5.7 Cyclic Redundancy Check (CRC) 268

Anhang 273

A.1 Inhalt der CD 273
A.2 Bezugsquellen 277
A.3 Literaturverzeichnis 281

Stichwortverzeichnis 283

Vorwort

Dieses Buch beschreibt die AVR-Mikrocontroller-Familie der Firma Atmel. Es gibt eine kurze Einführung in die Architektur der AVR-Mikrocontroller und beschreibt die Peripherie-Module und Eigenschaften dieser Controller. Großen Raum nimmt die Beschreibung des Befehlssatzes ein. Die Datenblätter aller in diesem Buch beschriebener AVR-Mikrocontroller sowie zahlreiches Informationsmaterial von der Atmel Home Page befinden sich als PDF-File auf der beiligenden CD.

Die zum Einstieg notwendigen Softwaretools, wie Assembler und AVR-Studio (Simulator) werden vorgestellt und befinden sich ebenfalls auf der CD. Der vorgestellte AVR-Programmer und die Experimentierplatine kann der interessierte Leser beim Elektronik Laden in Detmold (siehe Anhang) erwerben. Damit sollte der schnelle Einstieg in die Programmierung der AVR-Mikrocontroller keinerlei Schwierigkeiten bereiten. Um den Einstieg noch weiter zu vereinfachen, werden insgesamt sieben Anwendungen ausführlich besprochen. Die benötigten Sourcecodes sind abgedruckt und als Assembler-Dateien auf der CD abgelegt.

Für die freundliche Unterstützung bei der Erstellung des Manuskripts, der Programme und der Platinen bedanken wir uns bei den Firmen Beta-Layout und Cadsoft, bei Bob Henderson und Peter Jones von Atmel in England, bei Herrn Becker von Ineltek in Heidenheim, bei Herrn Yahya vom Ingenieurbüro Yahya in Erkelenz und Herrn Kausler von Hewlett-Packard in Böblingen.

Dem Elektor-Verlag sind wir für die Annahme und Veröffentlichung dieses Buches sehr dankbar. Insbesondere Bedanken wir uns bei Herrn Krings und Herrn Klein für die gute Zusammenarbeit und deren Geduld mit uns.

Deisenhofen, im Frühjahr 1999

Safinaz Volpe, Francesco P. Volpe

1 Einleitung

Bei der AVR-Mikrocontroller-Familie von Atmel handelt es sich um 8-Bit Mikrocontroller mit RISC-Architektur (Reduced Instruction Set Computer) für die unterschiedlichsten Anwendungen. Schon bei der Entwicklung der AVR-Mikrocontroller hat Atmel sehr eng mit Herstellern von Hochsprachen-Compilern zusammengearbeitet, um die Architektur und den Befehlssatz für den compilierten C-Code so effizient wie möglich zu gestalten Dieses Buch beschreibt die Architektur der AVR-Mikrocontroller, die Software- und Hardware-Entwicklungstools sowie sieben ausführlich beschriebene Anwendungen.

In Kapitel 2 wird eine kurze Einführung in die AVR-Mikrocontroller-Architektur gegeben. Es werden alle bis zur Drucklegung dieses Buches verfügbaren Mikrocontrollertypen dieser Familie mit ihren Eigenschaften angegeben. Anschließend werden die implementierten Peripherie-Module kurz vorgestellt. Abgeschlossen wird dieses Kapitel mit einer ausführlichen Erläuterung des AVR-Mikrocontroller-Befehlsatzes. Als Ergänzung und als weiterführende Literatur befinden sich die kompletten Datenblätter aller in diesem Buch vorgestellten AVR-Mikrocontroller als PDF-Datei auf der Begleit-CD.

Die zur Programmentwicklung benötigten Software-Tools werden in Kapitel 3 vorgestellt. Der Atmel AVR-Assembler wird in zwei Version, eine für MS-DOS und eine für Microsoft Windows, ausgeliefert. Dieser Assembler wandelt die Mnemonics in Maschinencode um, der dann in die AVR-Mikrocontroller programmiert werden kann.

Zur vorherigen Simulation des Assembler-Programms hat Atmel einen Simulator angeboten. Dieser Simulator wird nun nicht mehr unterstützt und ist von dem Software-Tools AVR-Studio abgelöst worden. Mit dem AVR-Studio kann der Anwender sowohl das Programm simulieren, er kann aber auch, falls er einen AVR-Emulator AT90ICEPRO besitzt, zur Ansteuerung des Emulators hernehmen. Sowohl der AVR-Assembler, als das AVR-Studio-Simulations-/ Emulatorprogramm befinden sich auf der Begleit-CD.

Für den Einstieg mit neuen Mikrocontrollern ist ein dazugehöriges Programmiergerät unumgänglich. Damit der interessierte Leser ein einfaches und preiswertes Programmiergerät für die AVR-Mikrocontroller erhält, wird in Kapitel 4 ein solches Gerät samt der benötigten PC-Software vorgestellt. Diesen AVR-Programmer kann der Leser selber nachbauen oder beim Elektronik Laden in Detmold (siehe Anhang A.2) bestellen. Für die gehobenen Ansprüche bei der Programmentwicklung wird der AVR In-Circuit Emulator AT90ICEPRO vorgestellt. Für ein solches Hardware-Tool muß man zwar leider einige tausend DM bezahlen, aber es macht sich für den professionellen Entwickler sehr schnell bezahlt.

Nichts erklärt einen Sachverhalt besser als ein Beispiel. Aus diesem Grund sind in Kapitel 5 insgesamt sieben unterschiedliche Anwendungen aufgelistet. Jede Anwendung wird ausführlich beschrieben und der dokumentierte Assembler-Sourcecode wird angegeben. Dieser ist auch als Datei auf der Begleit-CD zu finden. Mikrocontroller bilden meist eine Schnittstelle zwischen dem Menschen und einem Gerät oder einer Maschine. Zu diesem Zweck müssen Mikrocontroller Signale empfangen und/oder senden. Mit anderen Worten: Die Beispielprogramme alleine reichen zum Ausprobieren nicht aus. Deshalb wird zu Beginn des Kapitels 5 eine AVR-Experimentierplatine beschrieben. Diese ist in der Lage, AVR-Mikrocontroller im PDIP-20-Gehäuse aufzunehmen. Alle in diesem Buch beschriebenen Anwendungen lassen sich mit der AVR-Experimentierplatine und ein paar anwendungsspezifischen Bauteilen realisieren.

2 Die AVR-Mikrocontroller-Familie

In diesem Kapitel wird die AVR-Mikrocontroller-Familie besprochen. Nach Auflistung aller z. Z. verfügbaren AVR-Mikrocontroller-Typen und deren Eigenschaften, wird eine Einführung in die AVR-Architektur gegeben. Anschließend werden die Adressierungsarten und kurz die Peripherie der unterschiedlichen AVR-Typen besprochen. Abgeschlossen wird Kapitel 2 mit einer ausführlichen Darstellung des kompletten Befehlssatzes. Die Datenblätter aller in diesem Kapitel beschriebener AVR-Mikroocontroller-Typen befinden sich als PDF-Dateien im Verzeichnis \Data auf der beiliegenden CD (siehe Anhang A.1).

2.1 Eigenschaften und Typen

Die wesentlichen Eigenschaften der AVR-Mikrocontroller-Familie sind:

- 89 Ein-Wort Befehle (AT90S1200) bzw. 118 bis 121 Ein-Wort Befehle
- Fast alle Befehle benötigen nur einen Maschinenzyklus (z. B. 125 ns bei 8 MHz)
- Taktfrequenz von DC bis maximal 12 MHz
- 16-Bit breite Befehle
- 8-Bit breiter Datenbus
- 512 Worte bis 64 kWorte internes FLASH für Programme

- 32 General Purpose Register (GPR)
- 3-Level Hardware Stack (AT90S1200) bzw. Stackpointer im SRAM
- Direkte-, Indirekte- und Relative-Adressierung
- Interner Analog-Komparator
- 8-Bit Timer/Counter mit programmierbaren 10-Bit Vorteiler
- 16-Bit Timer/Counter mit Capture Compare Mode
- 1 bis 3 PWM-Kanäle
- 10-Bit Analog/Digital-Wandler mit 8 Kanäle
- 5 bis 48 I/O-Leitungen
- Watchdog Timer mit chipinternen RC-Oszillator
- Programmierbare Sicherung gegen auslesen des Programmcodes

Die einzelnen Familienmitglieder und deren Eigenschaften faßt **Tabelle 2.1** zusammen.

Ganz grob kann man die einzelnen Typen in folgende Kategorien unterteilen: Der AT90S1200 ist das kleinste Mitglied der Familie. Er verfügt im Gegensatz zu allen übrigen AVR-Typen über einen Hardware-Stack und kein internes SRAM (Statisches Random Access Memory). Ferner kann der AT90S1200 den Speicher nur direkt adressieren. Der AT90S2313 hingegen ist eine deutliche Erweiterung zum AT90S1200. Er verfügt im Vergleich zum AT90S1200 über den doppelten Programm- und EEPROM-Speicher (Electrically Erasable and Programmable Read Only Memory) sowie über 128 Bytes SRAM, einen UART (Universal Asynchronous Receiver and Transmitter), einem 16-Bit Timer/Counter und einem PWM-Kanal (Pulse Width Modulator). Die Typen AT90S2323 und AT90S2343 sind in einem 8-poligen Gehäuse untergebracht. Sie sind ansonsten, abgesehen von der Peripherie, mit dem AT90S2313 vergleichbar. Die nächste Kategorie bilden die beiden Typen AT90S4414 und AT90S8515. Beide sind bis auf die Größe der Speicher identisch. Analoges gilt für die

	AT90S1200	AT90S2313	AT90S2323	AT90S2343	AT90S4414	AT90S4433	AT90S8515	AT90S8535	ATmega103	ATmega603
Befehle	89	120	120	120	120	118	120	118	121	121
Programmspeicher	1 K Bytes	2 K Bytes	2 K Bytes	2 K Bytes	4 K Bytes	4 K Bytes	8 K Bytes	8 K Bytes	128 K Bytes	64 K Bytes
EEPROM	64 Bytes	128 Bytes	128 Bytes	128 Bytes	256 Bytes	256 Bytes	512 Bytes	512 Bytes	4 K Bytes	2 K Bytes
SRAM	–	128 Bytes	128 Bytes	128 Bytes	256 Bytes	256 Bytes	512 Bytes	512 Bytes	4 K Bytes	4 K Bytes
General Purpose Register	32	32	32	32	32	32	32	32	32	32
Stack	3 Level	SRAM	SRAM	SRAM	SRAM	SRAM	SRAM	SRAM	SRAM	SRAM
I/O	15	15	5	5	32	32	32	32	48	48
max. Taktfrequenz	12 MHz	10 MHz	8 MHz	10 MHz	8 MHz	8 MHz	8 MHz	8 MHz	6 MHz	6 MHz
8-Bit Timer/Counter	1	1	1	1	1	1	1	1	2	2
16-Bit Timer/Counter	–	1	–	–	1	1	1	1	1	1
Capture/Compare Mode	–	ja	–	–	ja	ja	ja	ja	ja	ja
PWM	–	1	–	–	3	3	3	3	2	2
Analog-Comparator	ja	ja	–	–	ja	ja	ja	ja	ja	ja
UART	–	ja	–	–	ja	ja	ja	–	ja	ja
SPI	–	–	–	–	–	–	–	–	ja	ja
10-Bit ADC	–	–	–	–	–	–	–	–	8 Kanäle	8 Kanäle
RTC	–	–	–	–	–	–	–	–	ja	ja
Gehäuse	PDIP-20, SOIC-20	PDIP-20, SOIC-20	PDIP-8, SOIC-8	PDIP-8, SOIC-8	PDIP-40, PLCC-44, TQFP-44	PDIP-40, PLCC-44, TQFP-44	PDIP-40, PLCC-44, TQFP-44	PDIP-40, PLCC-44, TQFP-44	TQFP-64	TQFP-64

Tabelle 2.1: Die AVR-Mikrocontroller-Familie.

Typen AT90S4434 und AT90S8535. Im Gegensatz zu den Typen AT90S4414 und AT90S8515 besitzen sie einen 10-Bit Analog/Digital-Wandler mit acht Kanälen. Die letzte Kategorie bilden die Typen ATmega103 und ATmega603 mit 64 kWorte bzw. 32 kWorte.

2.2 AVR-Architektur

Die AVR-Mikrocontroller Familie weist Merkmale auf, die bei RISC (Reduced Instruction Set Computer) implementiert sind. Abweichend von anderen Controllern, die einen Bus für Befehle und Daten verwenden (von Neumann-Architektur), verwendet die AVR-Familie zwei getrennte Busse und Speicher für Befehle und Daten (Harvard-Architektur). Dadurch ist es möglich, verschieden breite Busse für Daten und Befehle zu benutzen. Die AVRs haben einen 8-Bit breiten Bus für Daten und einen 16-Bit breiten Bus für Befehle. Durch den breiteren Bus für die Befehle ist es möglich, fast alle Befehle als Ein-Wort Befehle zu implementieren. Durch die einstufige Pipeline kann, während ein Befehl ausgeführt wird, der nächste bereits aus dem Programmspeicher geholt werden. Der Programmzähler (PC, Program Counter) ist, je nach AVR-Mikrocontroller Typ und damit je nach Programmspeichergröße, zwischen 9- und 16-Bit breit und befindet sich im I/O-Adreßbereich (s. u.). Während eines Interrupts oder beim Aufrufen einer Unterprogrammroutine wird der Inhalt des PC auf den Stack abgelegt. Der AT90S1200 verfügt über einen 3-Level Hardware-Stack. Alle anderen AVR-Mikrocontroller Typen legen den Stack im SRAM ab und greifen über einen Stack-Pointer (SP) auf diesen zu. Der Stack-Pointer ist 16-Bit breit und befindet sich ebenfalls im I/O-Adreßraum.

Die einzelnen Typen der AVR-Familie unterscheiden sich in der Größe des Programmspeichers, der zwischen 512 x 16 Bit und 65536 x 16 Bit liegt, Größe des EEPROM-Datenspeichers (zwischen 64 Byte und 2 kByte), Größe des internen SRAM-Datenspeichers (zwischen 128 Byte und 4 kByte), der Anzahl der Ports und der implementierten Peripherie, wie Analog/Digital-Wandler, UART usw. Für die Peripherie sind insgesamt 64 I/O-Speicherstellen vorgesehen, die direkt über den OUT-Befehl angesprochen werden können. Alternative lassen sich diese I/O-Speicherstellen im Adreßraum $20 bis $5F des Datenspeichers ansprechen.

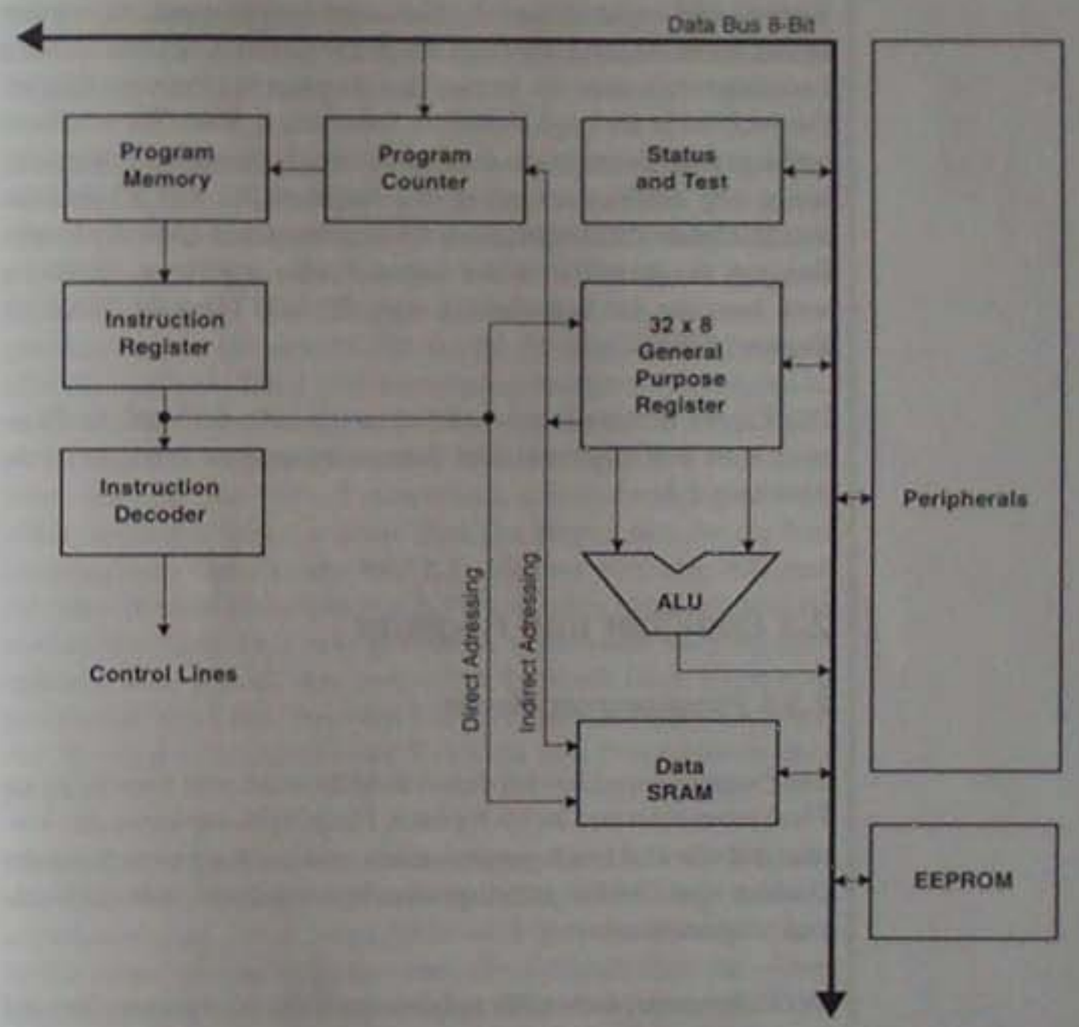


Bild 2.1: Architektur der AVR-Mikrocontroller Familie.

Der Controllerkern verfügt über eine 8-Bit breite ALU (Arithmetic Logic Unit), die zusammen mit den 32 Allzweckregister (GPR, General Purpose Register R0...R31), im folgenden einfach nur Register genannt, direkt verbunden ist. Diese 32 Register sind ungefähr das, was bei anderen Mikrocontrollern der Akkumulator ist. Alle 32 Register sind im Adreßraum des Datenspeichers gemappt (dazu später mehr). Beim AT90S1200 dient das letzte Register, bei den anderen Familienmitgliedern die letzten drei Register als Pointer (Zeiger). Die ALU ist in der Lage, Addition, Subtraktion, Shift (Bit schieben) und logische Operationen durchzuführen. So kann man beispielsweise eine Addition zwischen den Registern R1 und R2 und das anschließende Zurückspeichern des Ergebnisses in eines der beiden Register in einem Befehl mit einem Zyklus ausführen. Abhängig vom Ausgang der Verknüpfung setzt die ALU Flags im STATUS-Register (SREG).

Der Zugriff auf den Datenspeicherbereich (also auch auf die Register) kann mit insgesamt fünf Adressierungsarten erfolgen (siehe Abschnitt 2.3).

2.3 Speicher und Register

2.3.1 Programmspeicher

Der Programmspeicher bei der AVR-Mikroocntroller Familie ist als Flash ausgeführt und ist 16-Bit breit. Flash-Speicher bietet den Vorteil, daß sie elektrisch programmier- und auch elektrisch wieder löschbar sind. Der Programmspeicher ist mindestens 1000 mal lösch- und programmierbar.

Der Programmspeicher läßt sich bei den AVR-Mikrocontrollern auf zwei unterschiedliche Arten programmieren: Zum einen parallel, d. h. die zu programmierenden Daten werden parallel mit je 8 Bit an die Anschlüsse PB0 bis PB7 angelegt (siehe Datenblätter). Es besteht aber auch die Möglichkeit, den Programmspeicher seriell über eine SPI-Schnittstelle (Serial Peripheral Interface) zu programmieren. Damit ist es sogar möglich, die Mikrocontroller in der Schaltung (In-Circuit) zu programmieren. Dazu werden die Anschlüsse PB5 bis PB7 verwendet. Der AVR-Programmer, der in Kapitel 4 vorgestellt wird, verwendet außschließlich die serielle Programmierung.

Der Programmspeicher wird von der Adresse 0 linear bis zur Endadresse adressiert. Die Größe des Programmspeicher kann man aus **Tabelle 2.1** (Seite 11) entnehmen.

Interessant ist die Möglichkeit, Daten bzw. Konstanten in eine Tabelle abzulegen, die sich im Programmspeicher befindet. Das ist bei allen AVR-Mikrocontrollern außer beim AT90S1200 möglich. Das Auslesen der Daten wird mit dem LPM-Befehl (Load Program Memory) bzw. bei den ATmega103 aufgrund des größeren Programmspeichers mit dem ELPM-Befehl (Extended Load Program Memory) realisiert. Dazu geht man folgendermaßen vor: Die Adresse, an der die Daten im Programmspeicher stehen, muß in den Z-Pointer (R30 und R31) geladen werden. Man muß beachten, daß diese Adresse in den Bits 1 bis 15 stehen muß. Das niederwertigste Bit 0 wählt das untere bzw. das obere Byte des Wortes aus, das im Programmspeicher steht (siehe **Bild 2.2**). Hieraus wird klar, daß man die Adresse nicht einfach in den Z-Pointer laden kann, sondern sie erst um eine Stelle nach links geschoben werden oder mit zwei multipliziert werden muß, was zum selben Ergebnis führt. Führt man anschließen den LPM- bzw. den ELPM-Befehl aus, wird das Register R0 mit dem so adressierten Bytes aus dem Programmspeicher geladen.

Mit dem LPM-Befehl lassen sich somit insgesamt 32 kWorte adressieren, da der Z-Pointer in diesem Fall eine 15-Bit lange Adresse aufnehmen kann. Beim ATmega103 wird über den Z-Pointer eine 16-Bit lange Adresse aufgenommen. Die Auswahl über das untere oder obere Byte aus dem Programmspeicher wird über das Bit 0 im RAMPZ-Register (Adresse $3B im I/O- bzw. $5B im Datenspeicheradreßraum) vorgenommen. Beide Befehle werden in Abschnitt 2.9 ausführlich erklärt. In den Abschnitten 5.3 und 5.7 sind zwei Beispiele angegeben, bei denen der LPM-Befehl verwendet wird.

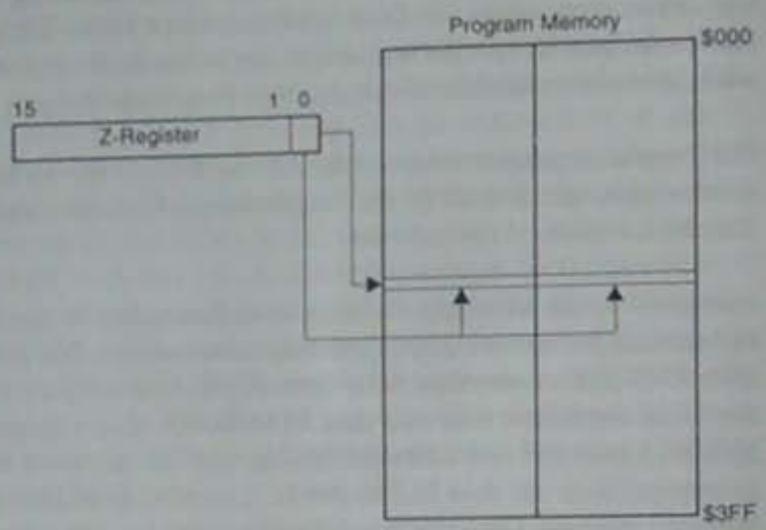


Bild 2.2: Laden eines Bytes aus dem Programmspeicher (hier 1 kWort groß).

2.3.2 Datenspeicher (SRAM)

Die AVR-Mikrocontroller verfügen über ein 128 Bytes bis 4 kBytes größes SRAM. Einzige Ausnahme bildet der AT90S1200, der über kein SRAM verfügt (siehe **Tabelle 2.1**). Dieses SRAM kann z. B. als Datenspeicher dienen. Es wird auch dazu verwendet, den Stack aufzunehmen (siehe Abschnitt 2.6). Die Adreßräume des Programm- und Datenspeichers sind, wie bereits erwähnt, getrennt. Der Adreßraum des Datenspeichers beginnt bei der Adresse 0 und endet bei der Adresse $FFFF (hexadezimal). Er umfaßt also 65536 Speicherstellen (siehe **Bild 2.3**).

Das SRAM selber beginnt aber erst ab der Adresse $60. Obwohl die Register R0 bis R31 und die 64 I/O-Speicherstellen direkt ansprechbar sind, die I/O-Speicherstellen z. B. über die Befehle IN und OUT, werden diese in den Datenspeicheradreßraum gespiegelt. Damit ist es möglich, die Register und die I/O-Speicherstellen wie ganz normalen Datenspeicher zu behandeln. Die Register R0 bis R31 befinden sich an den Adressen $0 bis $1F und die I/O-Speicherstellen an den Adressen $20 bis $5F im Adreßraum des Datenspeichers. Das interne SRAM schließt sich dann ab der Adresse $60 an. Die Größe des SRAM ist vom AVR-Mikrocontroller abhängig und kann aus der **Tabelle 2.1** entnommen werden.

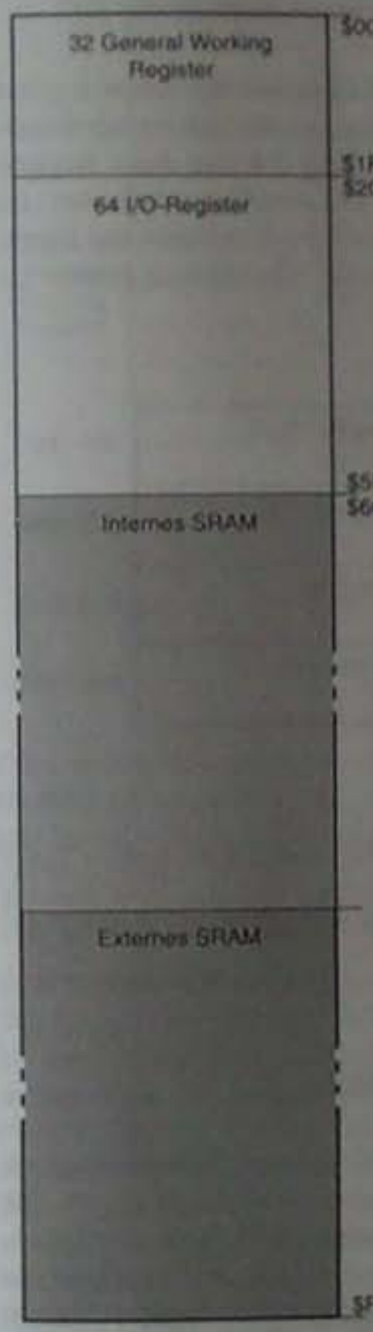


Bild 2.3: Der Adreßraum des Datenspeichers

Die Typen AT90S4414 und AT90S8515 sowie die Bausteine ATmega103/603 können zusätzlich externes SRAM ansprechen. Dieser externe Speicher kann nach der letzten Adresse des internen SRAM angeschlossen werden. Da die maximale Größe des internen SRAM 4 kBytes ist (siehe **Tabelle 2.1**) und der Adreßraum des Datenspeicher maximal 64 kBytes ist, können maximal 60 kBytes externer Datenspeicher adressiert werden.

2.3.3 EEPROM

Alle AVR-Mikrocontroller verfügen über ein internes EEPROM. Die Größe dieses EEPROMs kann man wieder der **Tabelle 2.1** entnehmen.

Das EEPROM befindet sich in einem von Programm- und Datenspeicher getrennten Adreßraum.

Das byteweise schreiben in das EEPROM sowie das Lesen aus dem EEPROM geschieht über spezielle Register, die sich im Adreßraum der I/O-Speicherstellen befinden. In **Bild 2.4** sind diese Register abgebildet. Neben den Adressen im I/O-Adreßraum befinden sich in Klammern die äquivalenten Adressen im Adreßraum des Datenspeichers. Es sei daran erinnert, daß die I/O-Adressen dorthin gespiegelt werden.

EEPROM Adress-Register (high) EEARH	$1F ($3F)
EEPROM Adress-Register (low) EEARL	$1E ($3E)
EEPROM Data-Register EEDR	$1D ($3D)
EEPROM Control-Register EECR	$1C ($3C)

Bild 2.4: Steuerregister zum byteweisen Schreiben und Lesen des EEPROMs.

Aufgrund der Größe des EEPROMs besitzen nur die beiden Typen ATmega103 und ATmega603 das Register EEARH. Alle anderen Typen kommen mit einem Adreßbyte zur Adressierung des EEPROMs aus.

2.3.4 General Purpose Register

Die AVR-Mikrocontroller besitzen 32 General Purpose Register (GPR), im folgenden einfach als Register bezeichnet (**Bild 2.5**). Alle 32 Register sind mit der ALU verbunden. Dadurch ist es möglich, daß alle registerorientierten Befehle einen dirkten Zugriff auf die Register besitzen und innerhalb eines einzigen Maschinenzyklus ausgeführt werden. Die Register können auch über den Adreßraum des

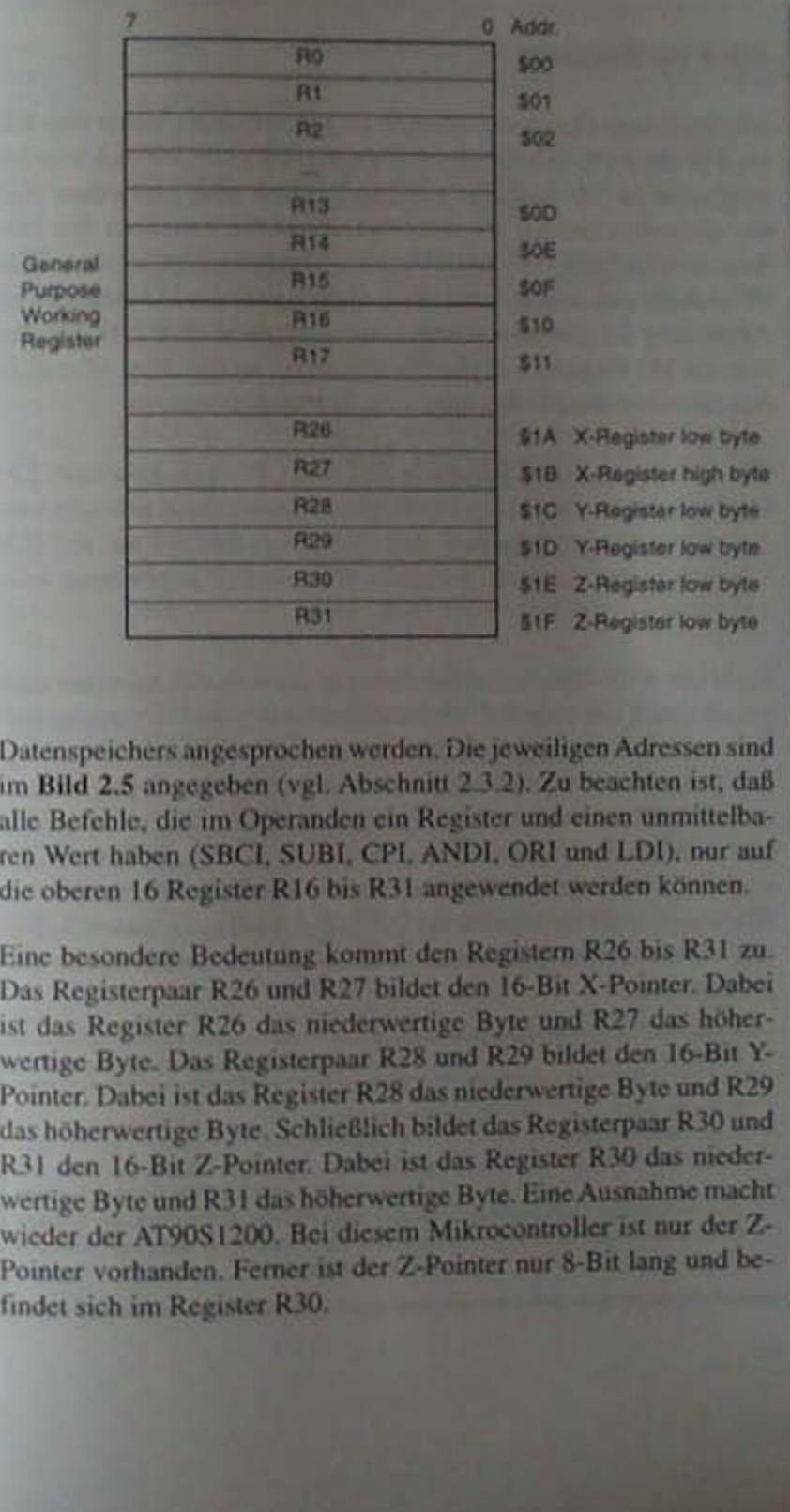


Bild 2.5: General Purpose Register

Datenspeichers angesprochen werden. Die jeweiligen Adressen sind im **Bild 2.5** angegeben (vgl. Abschnitt 2.3.2). Zu beachten ist, daß alle Befehle, die im Operanden ein Register und einen unmittelbaren Wert haben (SBCI, SUBI, CPI, ANDI, ORI und LDI), nur auf die oberen 16 Register R16 bis R31 angewendet werden können.

Eine besondere Bedeutung kommt den Registern R26 bis R31 zu. Das Registerpaar R26 und R27 bildet den 16-Bit X-Pointer. Dabei ist das Register R26 das niederwertige Byte und R27 das höherwertige Byte. Das Registerpaar R28 und R29 bildet den 16-Bit Y-Pointer. Dabei ist das Register R28 das niederwertige Byte und R29 das höherwertige Byte. Schließlich bildet das Registerpaar R30 und R31 den 16-Bit Z-Pointer. Dabei ist das Register R30 das niederwertige Byte und R31 das höherwertige Byte. Eine Ausnahme macht wieder der AT90S1200. Bei diesem Mikrocontroller ist nur der Z-Pointer vorhanden. Ferner ist der Z-Pointer nur 8-Bit lang und befindet sich im Register R30.

2.3.5 I/O-Register

Alle Ports und Peripherie-Module sind im I/O-Adreßraum von $00 bis $3F der AVR-Mikrocontroller abgelegt. Es handelt sich also um insgesamt 64 I/O-Register. Die I/O-Register sind zum einen über die speziellen Befehle IN und OUT, die Daten zwischen den I/O-Registern und den 32 General Purpose Register austauschen, in dem I/O-Adreßraum adressierbar. Zum andern sind die I/O-Register im Adreßraum des Datenspeichers gespiegelt (siehe **Bild 2.3**). Möchte man die I/O-Register wie den Datenspeicher ansprechen, so muß zu den Adressen der I/O-Register $20 dazu addiert werden.

Über die speziellen I/O-Befehle SBI und CBI (vgl. Abschnitt 2.9) können einzelne Bits in den I/O-Registern gesetzt bzw. gelöscht werden. Es ist aber zu beachten, daß diese zwei Befehle nur auf I/O-Register an den Adressen zwischen $00 und $1F angewendet werden können.

Nicht alle AVR-Mikrocontroller haben an allen 64 I/O-Adressen auch physikalisch ein Register implementiert. Aufgrund der implementierten Peripherie sind mehr oder weniger I/O-Register notwendig. Das Datenblatt des jeweiligen AVR-Mikrocontroller-Typs gibt Aufschluß darüber, welche I/O-Adressen mit I/O-Registern belegt sind (siehe Verzeichnis \Data auf der beiligenden CD). In **Tabelle 2.2** sind alle I/O-Register aufgelistet. Die beiden AVR-Mikrocontroller ATmega103/603 besitzen für die I/O-Register teilweise andere Adressen, die Abweichungen sind in der Tabelle in der letzten Spalte aufgeführt.

Tabelle 2.2: I/O-Register

I/O-Adresse	Register	Funktion	Abweichung bei ATmega103/603
$3F	SREG	Status Register	
$3E	SPH	Stack Pointer High	
$3D	SPL	Stack Pointer Low	
$3C	reserviert		XDIV XTAL Divide Control Register
$3B	GIMSK	General Interrupt Mask Register	RAMPZ RAM Page Z Select Register
$3A	GIFR	General Interrupt Flag Register	EICR External Interrupt Control register

$39	TIMSK	Timer/Counter Interrupt Mask Register	EIMSK External Interrupt Mask Register
$38	TIFR	Timer/Counter Interrupt Flag Register	EIFR External Interrupt Flag Register
$37	reserviert	TIMSK	
$36	reserviert	TIFR	
$35	MCUCR	MCU General Control Register	
$34	reserviert	MCUSR MCU Status Register	
$33	TCCR0	Timer/Counter0 Control Register	
$32	TCNT0	Timer/Counter0 (8-Bit)	
$31	reserviert	OCR0 Timer/Counter0 Output Compare Register	
$30	reserviert	ASSR Asynchronous Mode Status Register	
$2F	TCCR1A	Timer/Counter1 Control Register A	
$2E	TCCR1B	Timer/Counter1 Control Register B	
$2D	TCNT1H	Timer/Counter1 High Byte	
$2C	TCNT1L	Timer/Counter1 Low Byte	
$2B	OCR1AH	Timer/Counter1 Output Compare Register A High Byte	
$2A	OCR1AL	Timer/Counter1 Output Compare Register A Low Byte	
$29	OCR1BH	Timer/Counter1 Output Compare Register B High Byte	
$28	OCR1BL	Timer/Counter1 Output Compare Register B Low Byte	
$27	reserviert	ICR1H	
$26	reserviert	ICR1L	
$25	ICR1H	T/C 1 Input Capture register High Byte	TCCR2 Timer/COunter2 Control Register
$24	ICR1L	T/C 1 Input Capture register Low Byte	TCNT2 Timer/Counter2 (8-Bit)
$23	reserviert		OCR2 Timer/Counter2 Output Compare Register
$22	reserviert		
$21	WDTCR	Watchdog Timer Control Register	
$20	reserviert		
$1F	EEARH	EEPROM Address Register High Byte	
$1E	EEARL	EEPROM Address Register Low Byte	
$1D	EEDR	EEPROM Data Register	
$1C	EECR	EEPROM Control Register	
$1B	PORTA	Data Register Port A	
$1A	DDRA	Data Direction Register Port A	
$19	PINA	Input Pins Port A	
$18	PORTB	Data Register Port B	
$17	DDRB	Data Direction Register Port B	
$16	PINB	Input Pins Port B	
$15	PORTC	Data Register Port C	
$14	DDRC	Data Direction Register Port C	
$13	PINC	Input Pins Port C	
$12	PORTD	Data Register Port D	
$11	DDRD	Data Direction Register Port D	
$10	PIND	Input Pins Port D	
$0F	SPDR	SPI I/O Data Register	
$0E	SPSR	SPI Status Register	
$0D	SPCR	SPI COntrol Register	
$0C	UDR	UART I/O Data Register	
$0B	USR	UART Status Register	

$0A	UCR	UART Control Register
$09	UBRR	UART Baud Rate Register
$08	ACSR	Analog Comparator Control and Status Register
$07	ADMUX	ADC Multiplexer Select Register
$06	ADCSR	ADC Control and Status Register
$05	ADCH	ADC Data Register High
$04	ADCL	ADC Data Register Low
$03	reserviert	PORTE Data Register Port E
$02	reserviert	DDRE Data Direction Register Port E
$01	reserviert	PINE Input Pins Port E
$00	reserviert	PINF Input Pins Port F

2.3.6 STATUS-Register

Das Status-Register (SREG) gibt Auskunft über den Satus der CPU (Central Process Unit) und der ALU. Es befindet sich an der I/O-Register-Adresse $3F bzw. im Datenspeicheradreßraum an Adresse $5F (**Bild 2.6**).

Bild 2.6: Status-Register (SREG).

Bit	7	6	5	4	3	2	1	0
$3F ($5F)	I	T	H	S	V	N	Z	C

Den einzelnen Bits sind sogenannte Flags zugeordnet, die folgende Bedeutung haben:

Flag	Bedeutung
I	Global Interrupt Enable
T	Bit Copy Storage
H	Half Carry Flag
S	Sign Flag, S = N XOR V für vorzeichenbehaftete Vergleiche
V	Zweierkomplement-Überlauf-Flag
N	Negative-Flag
Z	Zero-Flag
C	Carry-Flag

2.4 Peripherie

2.4.1 I/O-Ports

Die AVR-Mikrocontroller verfügen über eine unterschiedliche Anzahl an I/O-Leitungen. Die Anzahl der I/O-Leitungen reicht dabei von 5 beim AT90S2323/2343 bis 48 beim ATmega103/603 (siehe **Tabelle 2.1**). Einige Mikrocontroller verfügen über insgesamt 6 Ports, die alphabetisch nummeriert werden. In **Tabelle 2.3** ist aufgelistet, welcher AVR-Mikrocontroller-Typ über welche Ports verfügt.

Tabelle 2.3: Verfügbare Ports der AVR-Mikrocontroller.

Typ	Port A	Port B	Port C	Port D	Port E	Port F
AT90S1200	–	ja	–	7-Bit	–	–
AT90S2313	–	ja	–	7-Bit	–	–
AT90S2323/2343	–	5-Bit	–	–	–	–
AT90S4414/8515	ja	ja	ja	ja	–	–
AT90S4434/8535	–	ja	ja	ja	–	–
ATmega103/603	ja	ja	ja	ja	ja	ja

Alle Ports, bis auf Ausnahme des Port F, sind bi-direktionale Ports. Das heißt, sie dienen als Ein- und Ausgangsleitungen. Hingegen ist Port F nur als Eingang verwendbar.

Zur Definition, ob es sich bei den Ports A bis E um Ein- oder Ausgänge handelt, existiert für jeden dieser Ports ein Data Direction Register (DDR), das die Richtung jedes einzelnen Port-Pins angibt. Ferner kann bei jedem einzelnen Port-Pin getrennt ein Pull-Up-Widerstand zugeschaltet werden. Aus **Tabelle 2.4** ersieht man die Richtung und den Zustand des Pull-Up-Widerstands.

DDRXn	PORTXn	I/O	Pull-Up	Bemerkung
0	0	Eingang	nein	Hochohmig (Tri-state)
0	1	Eingang	ja	PXn treibt Strom über internen Pull-Up-Widerstand, falls am Eingang low liegt
1	0	Ausgang	nein	Aktiv low am Ausgang
1	1	Ausgang	nein	Aktiv high am Ausgang

X: Gibt Port A, B, C, D oder E an
n: Gibt die Pin-Nummer an (0 bis 7)

Tabelle 2.4: *Datarichtungsregister (DDR).*

Zu beachten ist, daß die einzelnen Port-Pins alternative Funktionen haben können. So kann ein AVR-Mikrocontroller einen externen Speicher ansteuern. Dazu wird der Speicher an Port A und Port C angeschlossen. Port A liefert dann gemultiplext die Daten und das untere Byte der Adresse. Port C liefert das obere Byte der Adresse. Andere Port-Pins werden für die vorhandene Peripherie wie UART oder SPI benötigt. Die alternative Funktionen der Port-Pins wird an der Stelle beschrieben, an der das entsprechende Peripherie-Modul erklärt wird.

2.4.2 Synchrone serielle Schnittstelle (SPI)

Die SPI-Schnittstelle (Serial Peripheral Interface) dient zur schnellen synchronen Datenübertragung. In **Bild 2.7** ist das Blockdiagramm der SPI-Schnittstelle zu sehen.

Die Datenübertragung geschieht über die Signale MOSI (Master Out Slave In), MISO (Master In Slave Out), SCK (Serial Clock) und bei mehreren Teilnehmer auf dem Bus noch zusätzlich mit /SS (Slave Select). Zum Anschluß der AVR-Mikrocontroller an eine SPI-Schnittstelle werden einige Pins des Port B verwendet, die dann eine alternative Funktion haben (siehe **Tabelle 2.5**).

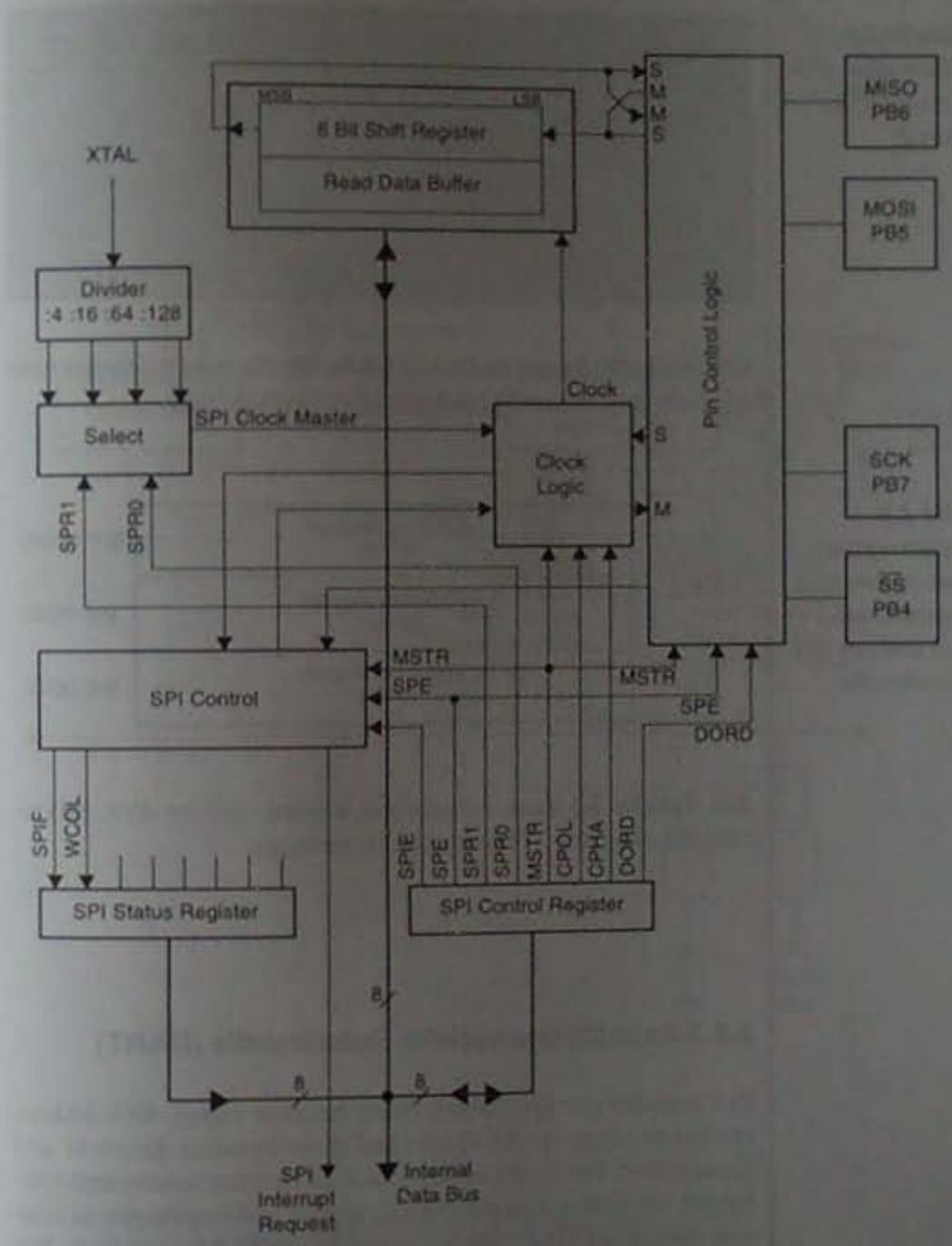


Bild 2.7: *Blockdiagram SPI-Schnittstelle.*

Tabelle 2.5: Alternative SPI-Belegung des Port B.

Signal	AT90S4414/4434/8515/8535	Atmega103/603
MOSI	PB5	PB2
MISO	PB6	PB3
SCK	PB7	PB1
/SS	PB4	PB0

Der Datentransfer von und zur SPI-Schnittstelle wird byteweise über spezielle Register im I/O-Adreßraum bewerkstelligt.

Bild 2.8: Steuerregister zum byteweisen Datentransfer von und zur SPI-Schnittstelle.

Aus **Tabelle 2.1** kann entnommen werden, welche AVR-Mikrocontroller über eine SPI-Schnittstelle verfügen.

2.4.3 Asynchrone serielle Schnittstelle (UART)

Zur asynchronen Datenübertragung besitzen einige AVR-Mikrocontroller einen UART (Universal Asynchronous Receiver and Transmitter). Der UART ermöglicht das byteweise senden und empfangen von Daten. Um gleichzeitig senden und empfangen zu können, besitzt der UART einen Transmitter (**Bild 2.9**) und einen Receiver (**Bild 2.10**), die unabhängig voneinander arbeiten.

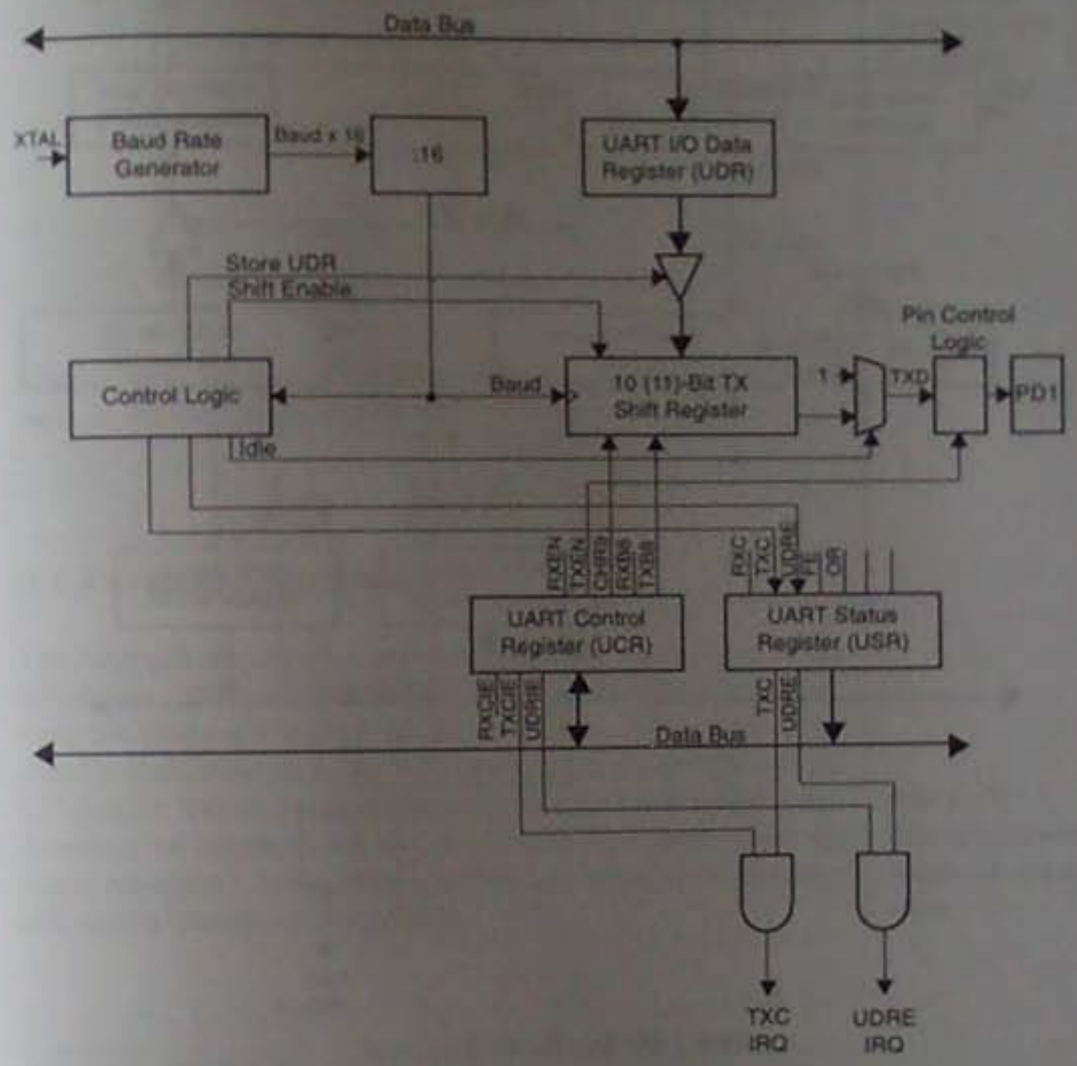


Bild 2.9: Der UART Transmitter.

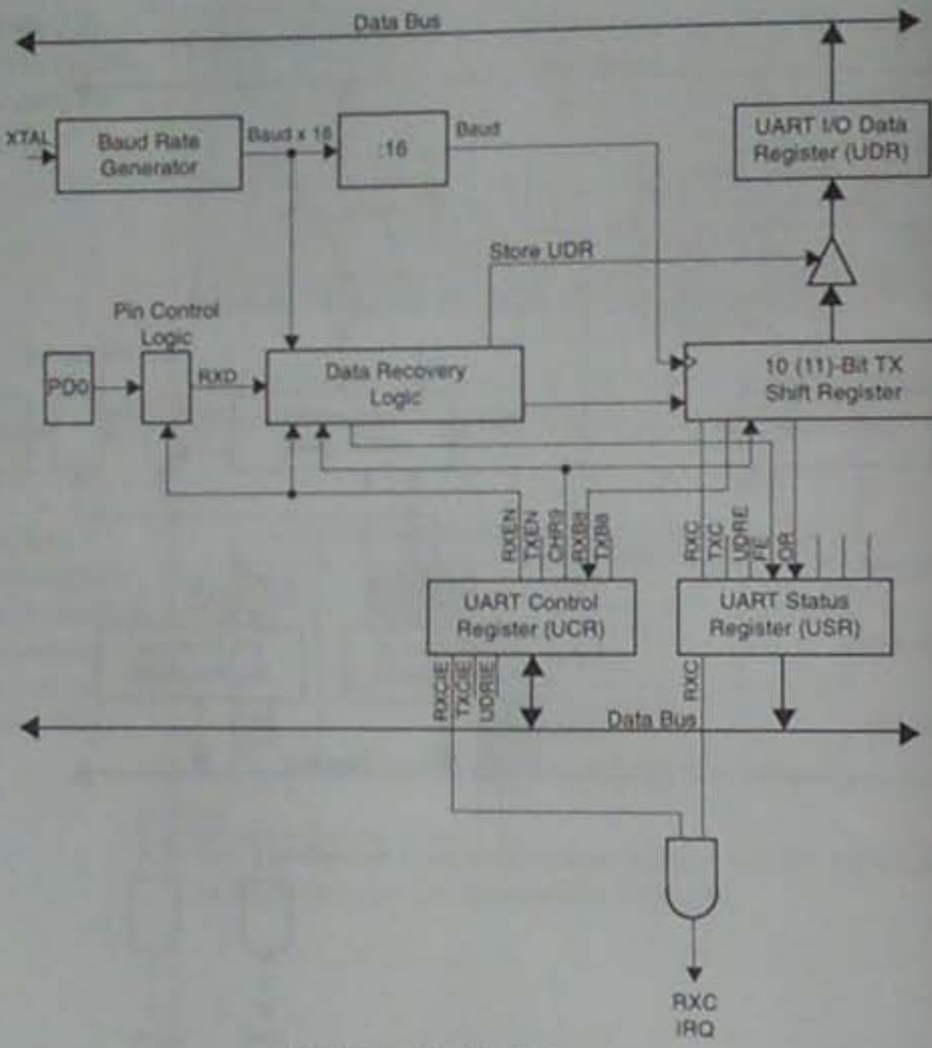


Bild 2.10: Der UART Receiver.

Die Baudratengenerierung übernimmt der UART selber. Dazu muß der Anwender nur die entsprechende Konstante in ein Steuerregister laden. Gesteuert wird der UART über vier spezielle Register im I/O-Adreßraum.

UART I/O Data Register UDR	$0C ($2C)
UART Status Register USR	$0B ($2B)
UART Control Register UCR	$0A ($2A)
UART Baud Rate Register UBRR	$09 ($29)

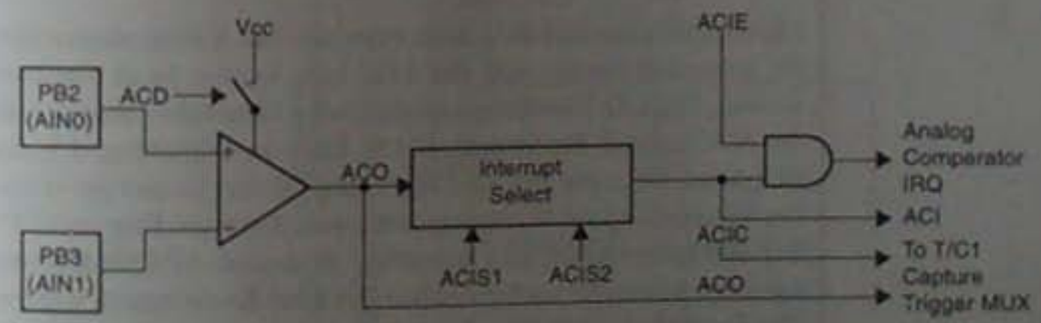
Bild 2.11: Steuerregister des UART.

Ein ausführliches Programmierbeispiel zur Ansteuerung des UART ist in Abschnitt 5.6 abgedruckt.

2.4.4 Analog-Komparator

Der Analog-Komparator vergleicht die Eingangsspannungen an den Eingängen AIN0 (AC+) und AIN1 (AC–) miteinander. Ist die Spannung am Eingang AIN0 größer als am Eingang AIN1, so wird der Analog-Komparatorausgang ACO (Analog Comparator Output) auf „1" gesetzt. Dieser Ausgang kann dazu benutzt werden den Timer/Counter1 zu triggern oder, um den Analog-Komparator-Interrupt (siehe Abschnitt 2.7) auszulösen. In **Bild 2.12** ist das Blockdiagramm des Analog-Komparator abgebildet.

Bild 2.12: Blockdiagramm Analog-Komparator.



Gesteuert wird der Analog-Komparator über das Analog Comparator Control and Status Register (ACSR) an Adresse $08 im I/O-Adreßraum bzw. an Adresse $28 im Datenspeicheradreßraum.

Die alternative Pinbelegung für die unterschiedlichen AVR-Mikrocontroller kann man aus **Tabelle 2.6** entnehmen.

Tabelle 2.6: Alternative Pinbelegung für den Analog-Komparator.

Typ	AIN0 (AC+)	AIN1 (AC–)
AT90S1200/2313	PB0	PB1
AT90S4414/4434/8515/8535	PB2	PB3
ATmega103/603	PE2	PE3

2.4.5 Analog/Digital-Wandler

Die AVR-Mikrocontrollertypen AT90S4434/8535 und ATmega103/603 besitzen einen Analog/Digital-Wandler (ADC, Analog Digital Converter) mit einer Auflösung von 10-Bit. Ferner verfügt der ADC über einen 8-Kanal-Multiplexer, mit dem acht Eingänge auf den eigentlichen ADC geleitet werden können (**Bild 2.13**).

Gesteuert wird der ADC über die vier Register ADMUX, ADCSR, ADCH und ADCL im I/O-Adreßraumm (**Bild 2.14**).

Mit dem Register ADMUX wird einer der acht Kanäle ausgewählt, der gewandelt werden soll. Der ADC kann in zwei Modi betrieben werden: Die A/D-Wandlung wird durch den Anwender gestartet oder der ADC wandelt kontinuierlich. Das Ende der Wandlung, d. h. der Zeitpunkt, wenn ein analoges Signal digitalisiert ist und zur weiteren Verarbeitung zur Verfügung steht, wird durch ein Flag im ADC-Status-Register (ADCSR) angezeigt. In diesem ADCSR-Register kann der Anwender auch zwischen den zwei Betriebsmodi wählen. Das Ergebnis der A/D-Wandlung steht dann in den Registern ADCH (Bit 8 und 9) und ADCL (Bit 0 bis 7).

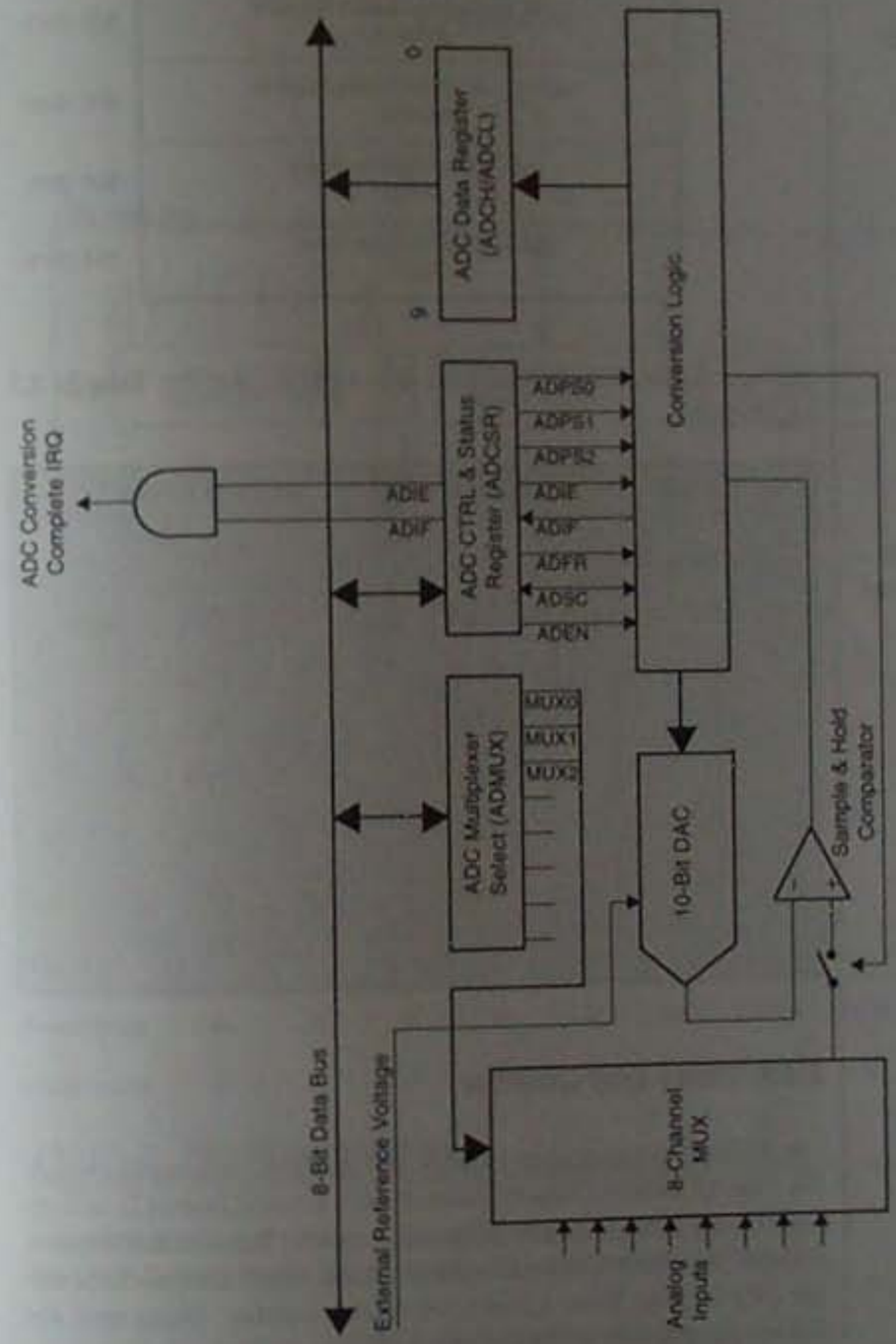


Bild 2.13: Blockdiagramm des 10-Bit ADC mit 8 Eingänge.

Bild 2.14: Steuerregister des ADC

ADC Multiplexer Select Register ADMUX	$07 ($27)
ADC Control and Status Register ADCSR	$06 ($26)
ADC Data Register (high) ADCH	$05 ($25)
ADC Data Register (low) ADCL	$04 ($24)

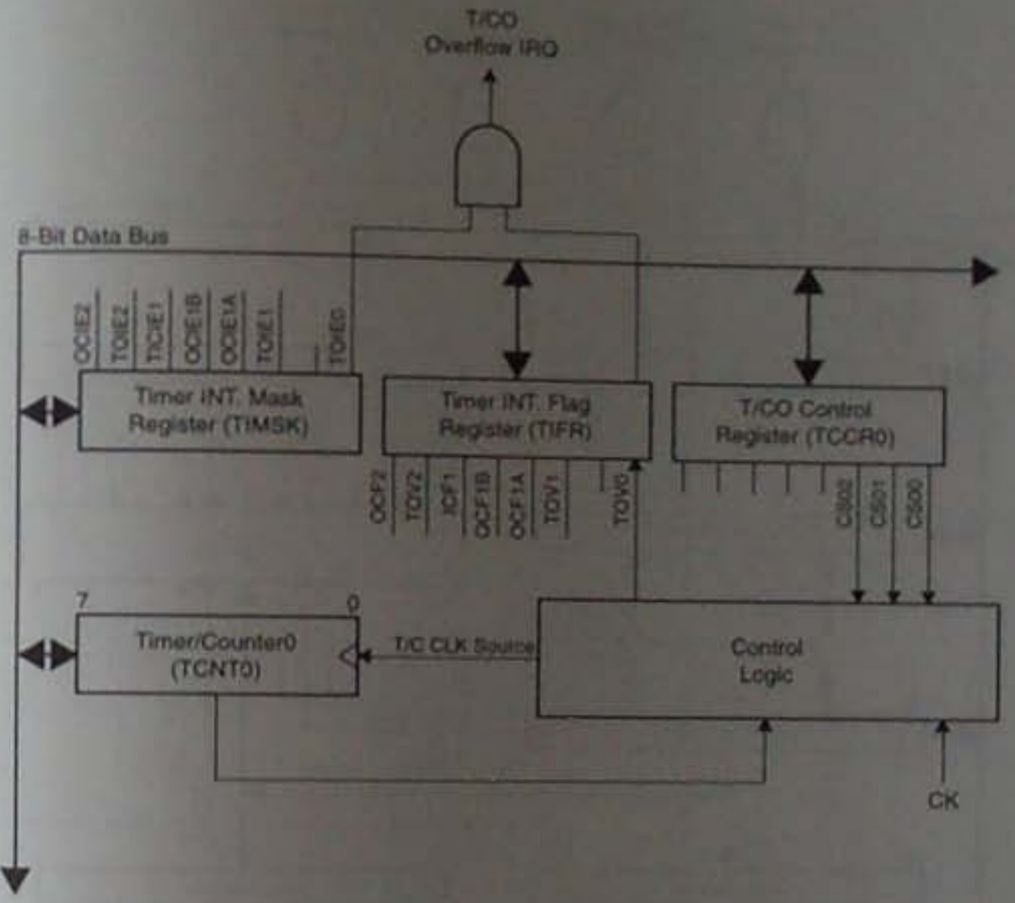
Die alternative Port-Beleung für den ADC ist aus der **Tabelle 2.7** ersichtlich.

Tabelle 2.7: Alternative Port-Belegung für den Analog/Digital-Wandler

Signal	AT90S4434/8535	Atmega103/603
ADC0	PA0	PF0
ADC1	PA1	PF1
ADC2	PA2	PF2
ADC3	PA3	PF3
ADC4	PA4	PF4
ADC5	PA5	PF5
ADC6	PA6	PF6
ADC7	PA7	PF7

2.4.6 Timer und Counter

Die AVR-Mikrocontroller-Familie verfügt über insgesamt zwei 8-Bit Timer/Counter (Timer/Counter0 und Timer/Counter2) und einem 16-Bit Timer/Counter (Timer/Counter1). Bei Controllertypen, die über den Timer/Counter2 verfügen, kann dieser auch als Echtzeituhr (RTC, Real Time Clock) verwendet werden. Dazu muß ein Schwingquarz mit 32768 Hz angeschlossen werden.



Bit	7	6	5	4	3	2	1	0	
$33 ($53)	–	–	–	–	–	CS02	CS01	CS00	TCCR0
Read/Write	R	R	R	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

• Bits 7...3 - Res: Reservierte Bits
Diese Bits sind für den AT90S4434/8535 reserviert und geben immer den Wert 0 aus

• Die Bits 2, 1 und 0 - CS02, CS01, CS00: Clock Select0, Bit 2, 1 und 0
Die Clock Select0 Bits 2, 1, 0 definieren die Vorteilerfaktor von Timer0

Bild 2.15: 8-Bit Timer/Counter0.

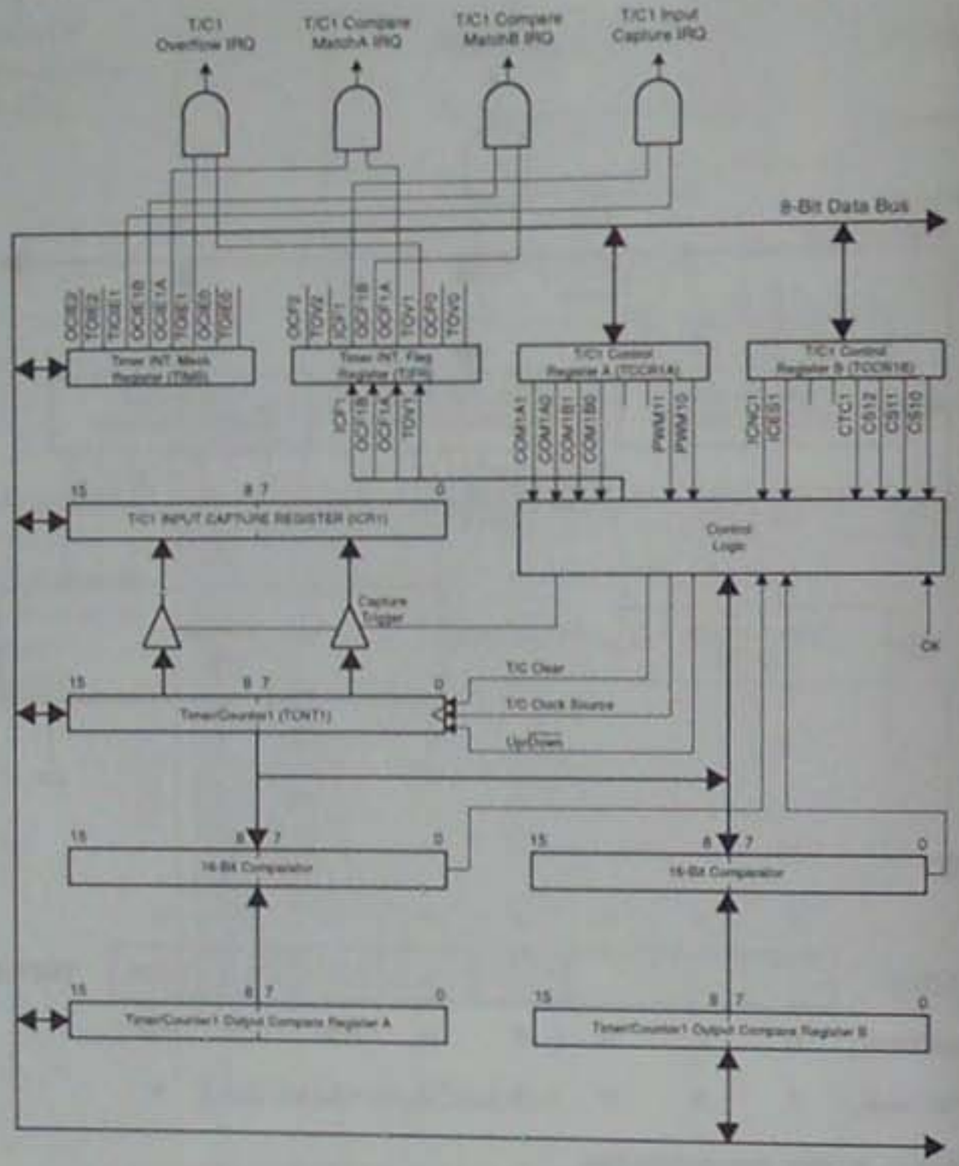


Bild 2.16: 16-Bit Timer/Counter1 mit Capture/Compare-Logik.

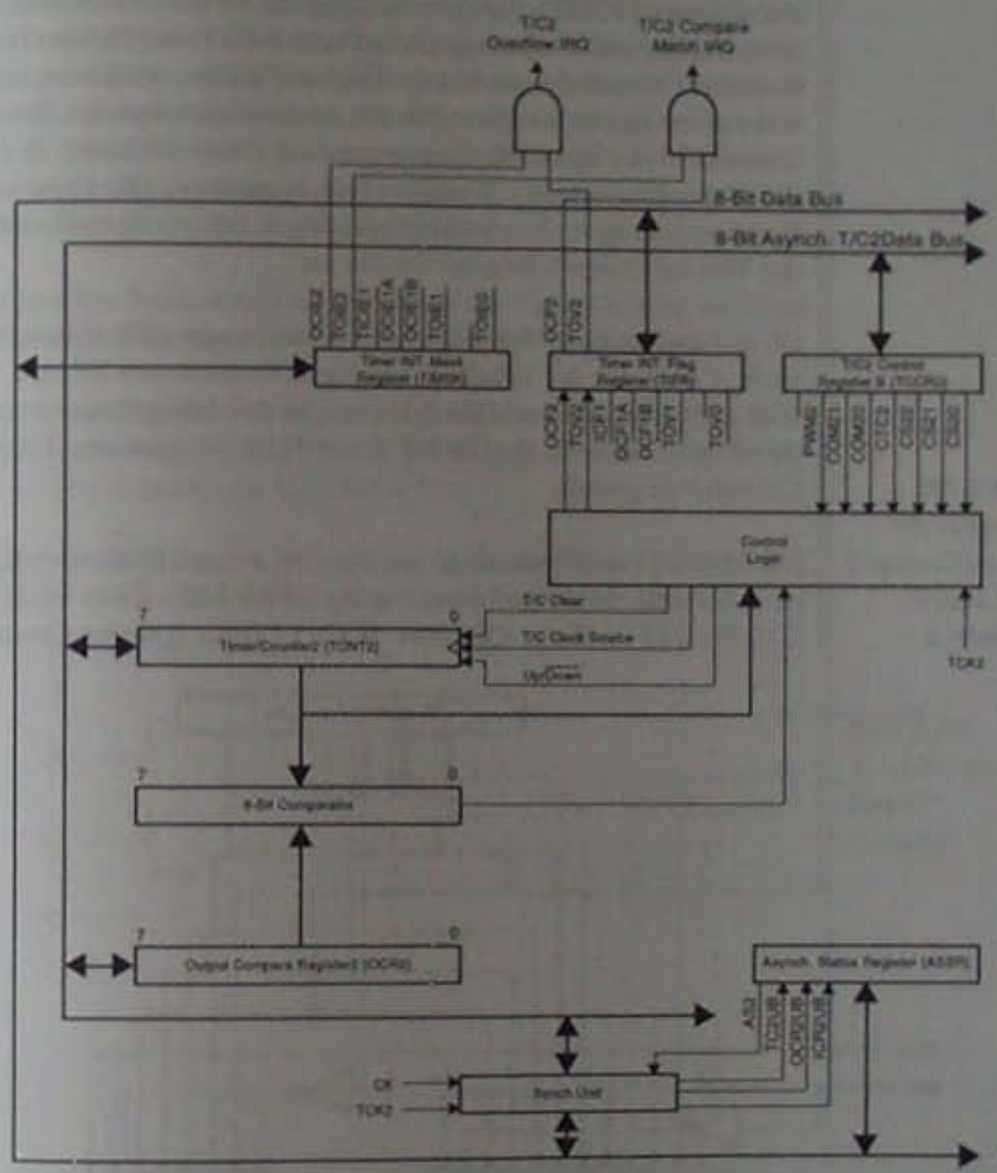


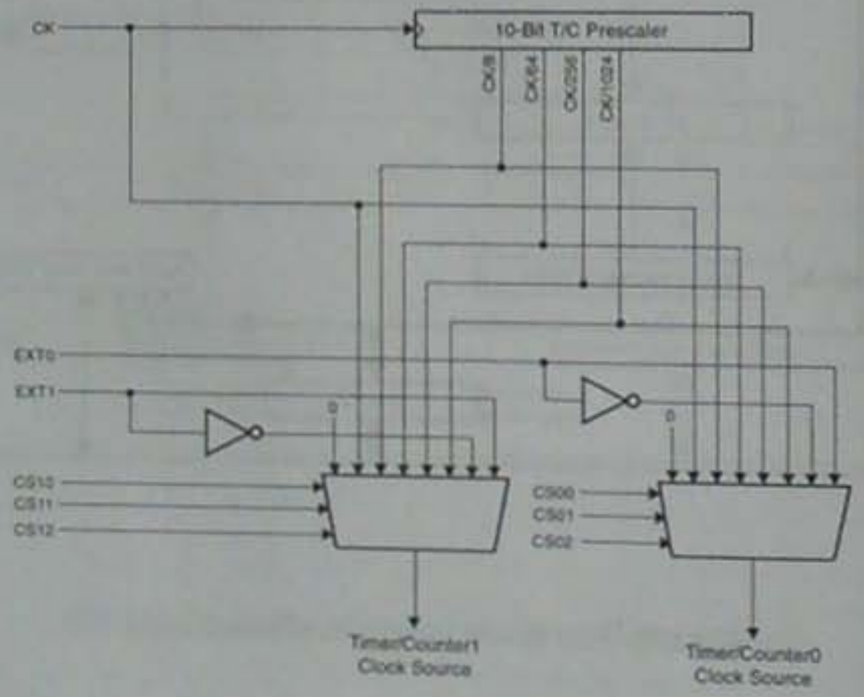
Bild 2.17: 8-Bit Timer/Counter2 mit Compare-Logik.

Bei einigen AVR-Mikrocontrollern verfügt der 16-Bit Timer/Counter1 über eine Capture/Compare-Logik und beim 8-Bit Timer/Counter2 nur über eine Compare-Logik. Mit der Capture-Funktion wird beim Auftreten einer Flanke an einem Pin der momentane Inhalt des Timer/Counter1 in ein Capture-Register gespeichert. Damit läßt sich z. B. der Zeitpunkt eines externen Signalwechsels bestimmen. Die Compare-Funktion dient dazu, einen Interrupt auszulösen, falls ein Timer/Counter den Wert im Compare-Register erreicht hat.

Ob und über welchen Timer/Counter ein bestimmter AVR-Controller verfügt, kann aus der **Tabelle 2.1** (Seite 11) entnommen werden. In **Bild 2.15**, **Bild 2.16** und **Bild 2.17** sind die Blockdiagramme des 8-Bit Timer/Counter0, des 16-Bit Timer/Counter1 und des Timer/Counter2 dargestellt.

Bild 2.18: Vorteiler für Timer/Counter0 und Timer/Counter1.

Dem 8-Bit Timer/Counter0 ist ein Vorteiler vorgeschaltet, der über eine Auswahl des Teilerfaktors verfügt (**Bild 2.18**). Über die Bits CS0 bis CS2 im Kontrollregister TCCRx (Timer Counter Control



Register) wird das Teilerverhältnis zwischen CK/8, CK/64, CK/256 und CK/1024 des Controller-Takts eingestellt. Sollte auf dem AVR-Mikrocontroller der 8-Bit Timer/Counter0 und der 16-Bit Timer/Counter1 implementiert sein, so verfügen beide über denselben Vorteiler. Das Teilerverhältnis kann aber für beider Timer/Counter unabhängig voneinander gewählt werden, da die Auswahllogik für das Teilerverhältnis doppelt ausgeführt ist.

Neben den Teilerverhältnissen können die Timer/Counter noch von einem externen Signal an einem Pin getaktet werden.

Der Timer/Counter2 verfügt über einen eigenen Vorteiler mit den Teilerverhältnisse TCK2/8 bis TCK2/1024 (siehe **Bild 2.19**). Dabei ist TCK2 ebenfalls der Controller-Takt.

In Abschnitt 5.1 ist im Programmbeispiel der Timer/Counter0 verwendet worden, um ein Ausgangssignal einer bestimmten Dauer zu erzeugen.

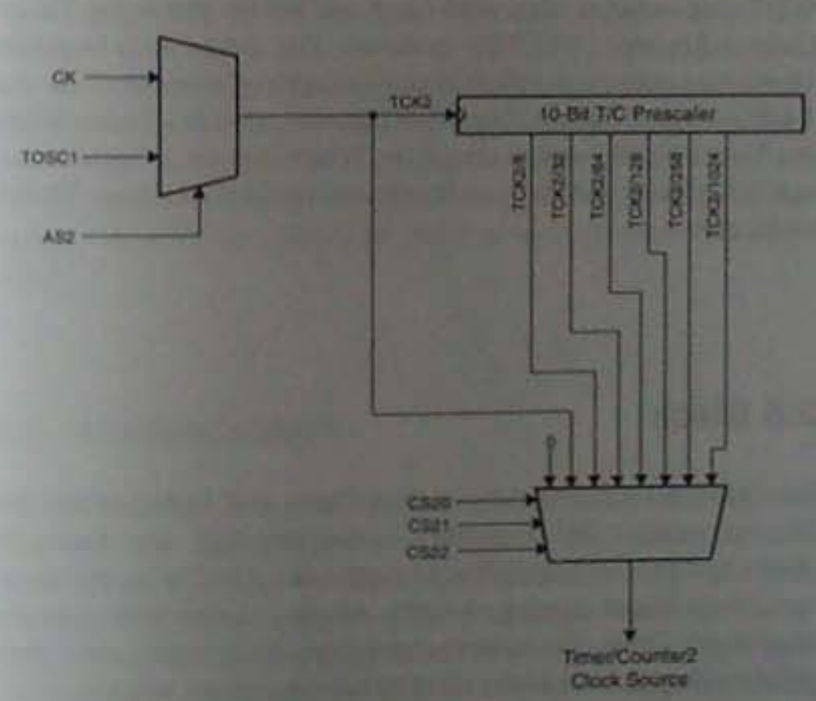


Bild 2.19: Vorteiler für Timer/Counter2.

Einige Timer/Counter-Module können im PWM-Modus (Pulse Width Modulator) betrieben werden. Damit ist es möglich, ein pulsweitenmoduliertes Signal zu erzeugen. Ein Beispiel dafür ist das Programm in Abschnitt 5.6. Die Anzahl der verfügbaren PWM-Kanäle bei den einzelnen AVR-Mikrocontrollern ist der **Tabelle 2.1** auf Seite 11 zu entnehmen. Die Timer/Counter-Module werden über Register im IO-Adreßraum gesteuert (siehe **Tabelle 2.2**, Seite 20...22).

2.4.7 Watchdog-Timer

Watchdog-Timer (WDT), siehe **Bild 2.20** auf Seite 39, werden dazu verwendet, um zu verhindern, daß sich Mikrocontroller „aufhängen". Das geschieht, indem ein WDT nach einer fest vorgegeben Zeit im Mikrocontroller einen Reset auslöst und so wieder einen definierten Zustand herstellt. Der WDT wird von einem unabhängigen internen Oszillator mit 1 MHz getaktet. Zum Betrieb werden keinerlei externe zeitbestimmenden Bauelemente benötigt. Ist der WDT eingeschaltet, dies wird durch ein Bit im Watchdog-Timer-Control-Register (WDTCR) gesteuert, löst dieser nach ungefähr 16 ms (das entspricht 16000 Maschinenzyklen bei einem Takt von 1 MHz) einen Reset aus. Diese Zeit läßt sich durch das Nachschalten des Vorteilers (Prescaler) erreichen. Wählt man beim Vorteiler einen Teilerfaktor von 2048, so lassen sich die 16 ms auf etwa 2,048 s verlängern.

2.5 Stack

Der Stack ist ein Speicher, in dem Daten und insbesondere der Programmzähler (PC, Program Counter) abgelegt wird. Dabei ist dieser Speicher als Stapel (Stack) organisiert. Alle Daten, die zuerst auf diesen Stapel abgelegt werden, rutschen immer tiefer mit neu abgelegten Daten. Die zuletzt abgelegten Daten liegen ganz oben auf dem Stapel und können als erste wieder gelesen werden.

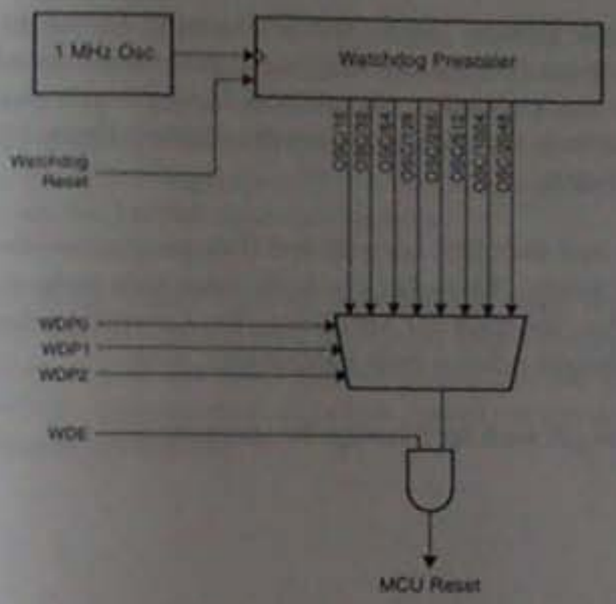


Bild 2.20: Watchdog-Timer mit Vorteiler.

Bei der AVR-Mikrocontroller-Familie müssen zwei unterschiedliche Arten von Stacks unterschieden werden. Zum einen der Hardware-Stack, der beim AT90S1200 implementiert ist (siehe Abschnitt 2.5.1). Zum anderen der Stack im Datenspeicher, wie er bei allen anderen AVR-Mikrocontrollern implementiert ist (siehe Abschnitt 2.5.2).

2.5.1 Hardware-Stack

Der AT90S1200 verfügt als einziger AVR-Mikrocontroller über einen Hardware-Stack. Dieser ist als 3-Level-Stack ausgeführt. Ein Stack-Pointer (SP), der auf den Anfang des Stacks zeigt, wird nicht benötigt. Ein Unterprogrammaufruf (rcall) kopiert den um eins erhöhten Inhalt des Programmzählers in die oberste Ebene des Stacks. Der Inhalt der obersten Ebene des Stacks wird vorher in die nächst

tiefere Ebene geladen. Nach einem Rücksprung aus einem Unterprogramm wird der Programmzähler mit dem Inhalt der obersten Ebene des Stacks geladen. Alle anderen Stack-Ebenen rücken um eine Ebene hoch. Der Inhalt der untersten (dritten) Ebene bleibt dabei unverändert.

Der Stack läuft über, falls mehr als drei Unterprogrammaufrufe verschachtelt werden. Wenn dies geschieht, kann nicht mehr sichergestellt werden, daß nach der Ausführung des Unterprogramms wieder zur richtigen Adresse zurückgesprungen wird.

Das gleiche gilt auch für Interrupt-Routinenaufrufe.

2.5.2 Stack im Datenspeicher

Mit Ausnahme des AT90S1200 liegt bei allen AVR-Mikrocontrollern der Stack im Datenspeicher. Ein Stack-Pointer zeigt dabei auf den Stack. Bevor irgendein Unterprogramm angesprungen wird, muß der Stack-Pointer initialisiert werden. Dabei legt man üblicherweise die höchste Stelle im Datenspeicher als den Anfang (oberste Ebene) des Stacks fest. Die Initialisierung des Stack-Pointers sieht folgendermaßen aus:

```
ldi     SPL,LOW(RAMEND) ;lade SP low mit RAMEND(low)
ldi     SPH,HIGH(RAMEND);lade SP high mit RAMEND(high)
```

In diesem Beispiel ist angenommen worden, daß der Stack-Pointer 16-Bit lang ist. In der ersten Zeile wird das niederwertige Byte des Stack-Pointers mit dem niederwertigen Byte der obersten Speicherstelle des Datenspeichers geladen. In der zweiten Zeile wird das höherwertige Byte des Stack-Pointers mit dem höherwertigen Byte der obersten Speicherstelle des Datenspeichers geladen.

Nach jedem Aufruf eines Unterprogramms wird der Stack-Pointer automatisch um zwei dekrementiert und nach jedem Rücksprung aus dem Unterprogramm automatisch um zwei inkrementiert. Das geschieht deshalb, weil der Programmzähler mehr als 8-Bit breit ist, der Datenspeicher hingegen nur 8-Bit breit ist. Es werden also zwei Speicherstellen im Datenspeicher benötigt.

Mit dem PUSH-Befehl kann ein Byte auf dem Stack abgelegt, mit dem POP-Befehl kann ein Byte vom Stack geholt werden. Bei diesen Befehlen wird der Stack-Pointer jeweils nur um eins dekrementiert bzw. inkrementiert, da es sich jeweils nur um ein Byte und nicht zwei Bytes handelt.

2.6 Reset

Die AVR-Mikrocontroller-Familie verfügt über eine interne Power-On-Reset (POR) Schaltung. Für die meisten Anwendngen genügt es, den /Reset-Eingang auf die positive Versorgungsspannung V_{CC} zu legen oder offen zu lassen, da dieser Eingang bereits intern über einen Widerstand auf V_{CC} liegt. Wenn dieser Eingang beim Einschalten auf high liegt, fängt ein interner Timer an zu zählen. Dieser Timer wird vom Watchdog-Timer getaktet. Erst wenn der Endstand erreicht ist, wird der interne Reset ausgeführt. Die zeitliche Verzögerung von typisch 16 ms erlaubt es dem angeschlossenen Quarz-Oszillator sich zu stabilisieren.

Wird ein zusätzlicher Reset benötigt, muß der Eingang /Reset beschaltet werden. Am einfachsten geschieht dies durch eine externe RC-Kombination am /Reset-Eingang (siehe **Bild 2.21**).

Die Zeitkonstante R • C des RC-Gliedes bestimmt, wann der /Reset-Eingang auf high gelegt wird. Dabei sollte der Widerstand R kleiner als 50 kΩ gewählt werden, da sonst die Spannung am /Reset-Ein-

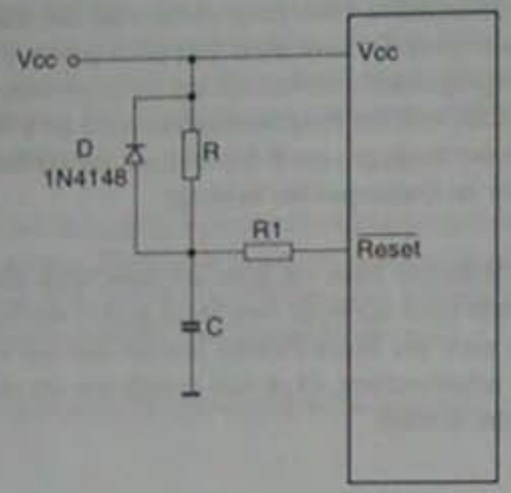


Bild 2.21: Externe Reset-Beschaltung.

gang zu klein ist. Die Diode D (1N4148) sorgt dafür, daß sich der Kondensator schnell entladen kann, wenn die Spannung ausgeschaltet wird. Der Widerstand R1 verhindert, daß bei voll aufgeladenem Kondensator ein zu hoher Strom in den /Reset-Eingang fließt. Der Wert für R1 sollte zwischen 100 Ohm und 1 KOhm liegen.

Nach jedem Reset beginnt die Programmausführung an der Adresse 0x00 im Programmspeicher (siehe Abschnitt 2.7).

2.7 Reset und Interrupt-Vektoren

Die ersten Stellen im Programmspeicher werden als Reset- und Interrupt-Vektoren definiert. Wird z. B. ein Reset ausgelöst, wird der Reset-Vektor an der Programmspeicheradresse 0x00 angesprungen. An dieser Stelle steht dann üblicherweise ein Sprungbefehl zur der Stelle im Programmspeicher, die nach einem Reset ausgeführt werden soll. Alle Peripherie-Module besitzen ebenfalls Vektoren, die sogenannten Interrupt-Vektoren. Löst ein Peripherie-Modul einen Interrupt aus, wird zu dem entsprechenden Interrupt-Vektor gesprungen. Dort steht dann ein Sprungbefehl zur Routine (Interrupt-Handler), die diesen Interrupt behandeln soll.

Die Adressen der Interrupt-Vektoren für die verschiedenen AVR-Mikrocontroller können aus den Datenblättern, die sich als PDF-Datei auf der Begleit-CD befinden, entnommen werden.

2.8 Taktoszillator

2.8.1 Quarz-Oszillator

Als Takt für die AVR-Controller wird üblicherweise ein Quarz-Oszillator verwendet (**Bild 2.22**). Dazu wird ein Schwingquarz der gewünschten Frequenz zwischen den Anschlüssen XTAL1 und XTAL2 angeschlossen. Damit der Quarz sicher anschwingt, sind noch die Kondensatoren C1 und C2 erforderlich. Diese haben einen typischen Wert von 22 pF.

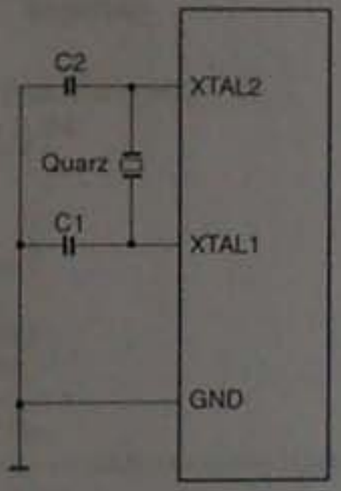


Bild 2.22: Beschaltung eines Quarz-Oszillators.

2.8.2 Interner RC-Oszillator

Der AT90S1200 kann über einen internen RC-Oszillator mit einem Takt von 1 MHz arbeiten. Dann kann der AT90S1200 ohne externe Bauteile arbeiten. Diese Betriebsart läßt sich über das Kontroll-Bit RCEN (RC Enable) im Flash-Speicher konfigurieren. Defaultmäßig ist diese Betriebsart ausgeschaltet. Sie läßt sich nur im parallelen Programmiermodus ändern.

2.8.3 Externer Takt

Soll ein externes Taksignal verwendet werden, das z. B. von einem TTL-Taktgenerator erzeugt wird, ist die Einspeisung nach **Bild 2.23** vorzunehmen.

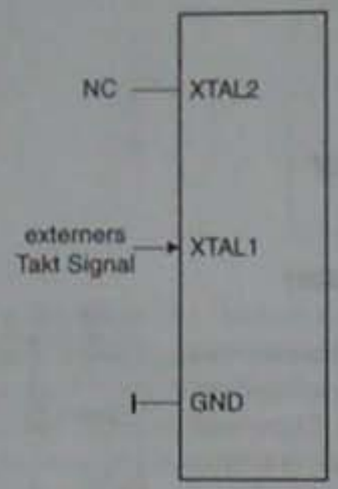


Bild 2.23: Externe Takteinspeisung.

2.9 Der Befehlssatz

Im folgenden wird der Befehlssatz der AVR-Mikrocontroller Familie besprochen. Dazu werden noch einige Definitionen erläutert:

A	Inhalt von A
A<7:0>	Bit 0 bis 7 von A
(A)	Inhalt der Speicherstelle, auf die der Inhalt von A zeigt (indirekte Adressierung)
/A	A ist invertiert zu nehmen
addr	Konstante Adreßdaten für Programmzähler
bit	Ein Bit eines Registers
K	Konstante oder Datenbyte (8 Bit)
mask	Maske für Bitmanipulation
offset	Offset zur Adresse
Port	I/O-Adresse
Rd	Ziel- und Quellregister im Register-File (GPR)
Rr	Quellregister im Register-File (GPR)
←	Zuweisung
.AND.	UND-Verknüpfung
.OR.	ODER-Verknüpfung
.XOR.	Exklusiv-ODER-Verknüpfung
SREG	STATUS-Register
C	Carry-Flag im STATUS-Register
Z	Zero-Flag im STATUS-Register
N	Negative-Flag im STATUS-Register
V	Zweierkomplement-Überlauf Flag
S	Signed-Flag, S = N .XOR. V (für vorzeichenbehaftete Vergleiche)
H	Half-Carry-Flag im STATUS-Register
T	Transfer-Bit im STATUS-Register (wird von den Befehlen BLD und BST benutzt)
I	Global Interrupt Enable/Disable Flag im STATUS-Register

Schattierte Flächen im STATUS-Register bedeuten, daß das Flag vom Ergebnis des Befehls beeinflußt werden kann. Im unten gezeigten Beispiel werden das Zero- und Carry-Flag beeinflußt.

I	T	H	S	V	N	Z	C

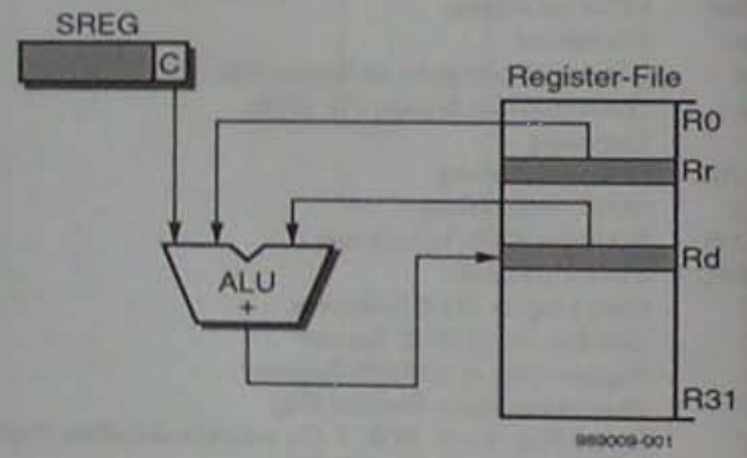
ADC Rd,Rr

ADC Rd,Rr — Addiere Register Rr mit Carry-Flag zum Register Rd.

Funktion: Rd ← Rd + Rr + C

Beschreibung: Der Inhalt des Registers Rr und das Carry-Flag werden zum Inhalt des Registers Rd addiert. Das Ergebnis der Addition steht im Register Rd. Der Inhalt des Registers Rr bleibt unverändert. Der Befehl ist für alle Register R0 bis R31 im Register-File zulässig.

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus, 1 Taktimpuls: 125 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

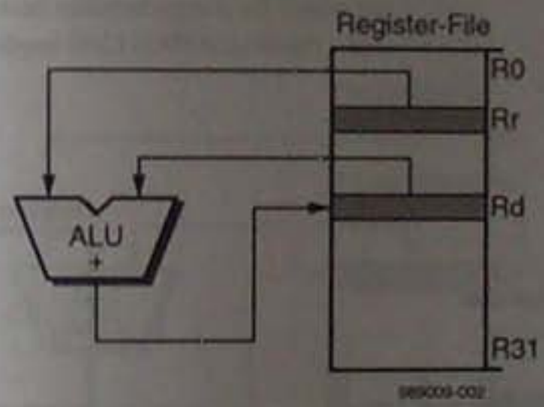
ADD Rd,Rr — Addiere Register Rr zum Register Rd.

Funktion: Rd ← Rd + Rr

Beschreibung: Der Inhalt des Registers Rr wird zum Inhalt des Registers Rd addiert. Das Ergebnis der Addition steht im Register Rd. Der Inhalt des Registers Rr bleibt unverändert. Der Befehl ist für alle Register R0 bis R31 im Register-File zulässig.

ADD Rd,Rr

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus, 1 Taktimpuls: 125 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

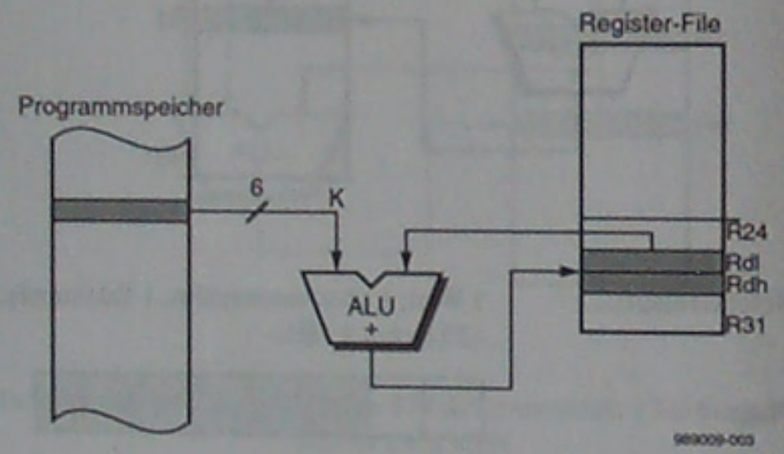
ADIW Rdl,K

ADIW Rdl,K — Addiere unmittelbaren Wert K zum Registerpaar Rdh:Rdl.

Funktion: Rdh:Rdl ← Rdh:Rdl + K

Beschreibung: Der unmittelbare Wert K (K im Bereich von 0 bis 63) wird zum Inhalt des Registerpaares Rdh:Rdl addiert. Dabei gibt Rdl das untere der beiden Register an. Gültige Werte für Rdl sind 24, 26, 28 und 30. Der Wert für Rdh ist dann automatisch 25, 27, 29 oder 31. Dieser Befehl bezieht sich also nur auf die oberen vier Registerpaare und ist somit für Zeigerbefehle bestens geeignet. (Nicht im AT90S1200 implementiert.)

Datenfluß:



Befehlsablauf: 1 Wort, 2 Maschinenzyklen, 2 Taktimpulse: 205 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

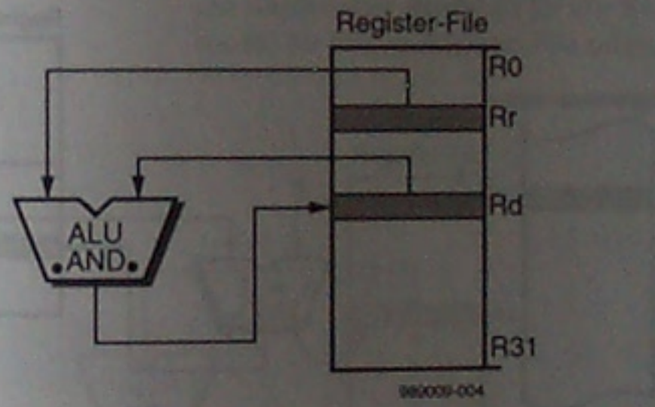
AND Rd,Rr

AND Rd,Rr — Register Rr und Rd logisch UND-verknüpfen.

Funktion: Rd ← Rd .AND. Rr

Beschreibung: Der Inhalt des Registers Rr wird mit dem Inhalt des Registers Rd logisch UND-verknüpft. Das Ergebnis der Verknüpfung steht im Register Rd. Der Inhalt des Registers Rr bleibt unverändert. Der Befehl ist für alle Register R0 bis R31 im Register File zulässig.

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus, 1 Taktimpuls: 125 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

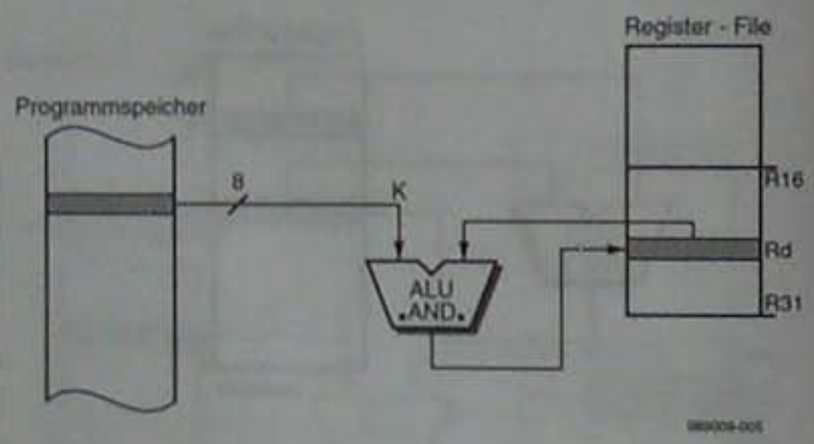
ANDI Rd,K

ANDI Rd,K — Unmittelbaren Wert K und Register Rd logisch UND-verknüpfen.

Funktion: Rd ← Rd .AND. K

Beschreibung: Der unmittelbare Wert K (K im Bereich von 0 bis 255) wird dem Inhalt des Regisers Rd logisch UND-verknüpft. Das Ergebnis der Verknüpfung steht im Register Rd. Der Befehl ist für die Register R16 bis R31 in der oberen Hälfte des Register-Files zulässig.

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus, 1 Taktimpuls: 125 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

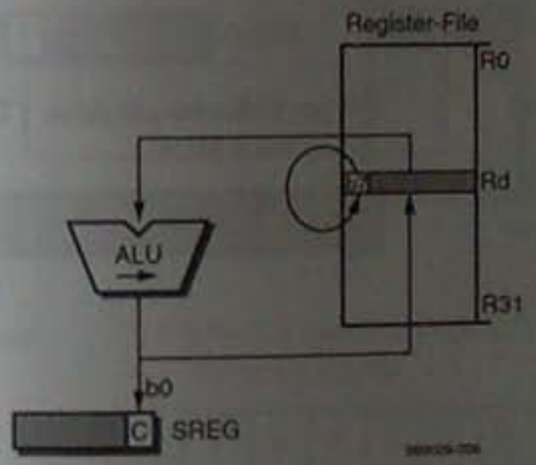
ASR Rd

ASR Rd — Schiebe das Register Rd arithmetisch nach rechts.

Funktion: Rd<7> ← Rd<7>; C ← Rd<0>; Rd<6> ← 0, Rd<5:0> ← Rd<6:1>

Beschreibung: Der Inhalt des Registers Rd wird um eine Stelle arithmetisch nach rechts geschoben. Dabei wird der Inhalt des Bit 7 erhalten, Bit 6 wird mit 0 überschrieben und der Inhalt des Bit 0 in das Carry-Flag übertragen. Dieser Befehl teilt eine Zahl, die in Zweierkomplement Schreibweise gespeichert ist, durch zwei. Dabei wird das Vorzeichen der Zahl nicht geändert. Das Carry-Flag kann zum Runden herangezogen werden. Der Befehl ist für alle Register R0 bis R31 im Register-File zulässig.

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus, 1 Taktimpuls: 125 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

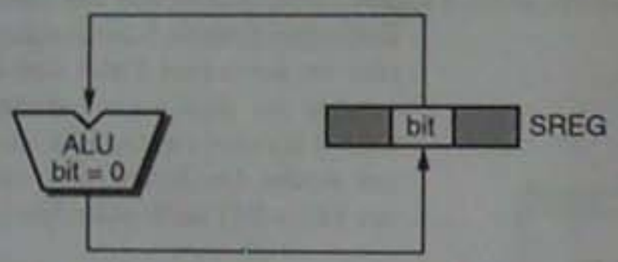
BCLR bit

BCLR bit — Lösche ein Bit im STATUS-Register.

Funktion: SREG<bit> ← 0

Beschreibung: Löscht das angegebene Bit im STATUS-Register. Der Wert bit kann im Bereich von 0 bis 7 liegen.

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus, 1 Taktimpuls: 125 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

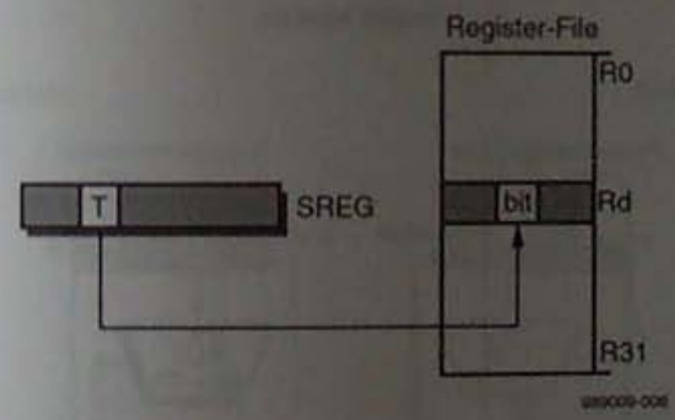
BLD Rd,bit

BLD Rd,bit — Lade T-Flag in das Register Rd.

Funktion: Rd<bit> ← T

Beschreibung: Das Trasfer-Bit aus dem STATUS-Register SREG wird in die Position bit des Registers Rd geladen. Der Zustand des Transfer-Bits wird nicht geändert. Mit dem Wert bit läßt sich jedes Bit im Register Rd adressieren. Der Befehl ist für alle Register R0 bis R31 imRegister-File zulässig.

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus, 1 Taktimpuls: 125 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

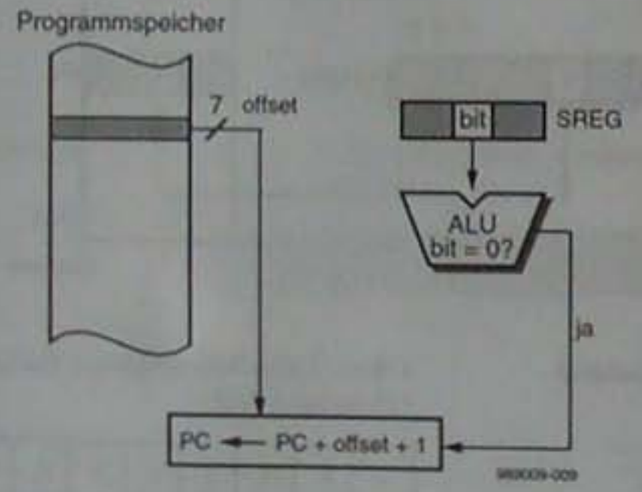
BRBC bit,offset

BRBC bit,offset — Springe relativ um den Wert offset falls bit im SREG gelöscht.

Funktion: PC ← PC + 1 + offset, falls SREG<bit> = 0

Beschreibung: Ein relativer Sprung um den Wert offset wird ausgeführt, falls ein einzelnes Bit des STATUS-Register SREG an der Position bit gelöscht ist. Mit dem Wert bit läßt sich jedes Bit im STATUS-Register adressieren. Der Wert offset wird als Zweierkomplement interpretiert, so daß relative Sprünge im Bereich von −64 bis +63 relativ zum Programmzähler PC ausgeführt werden können.

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus (2 bei Sprung), 1 Taktimpuls (2 bei Sprung): 125 ns (250 ns bei Sprung) bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

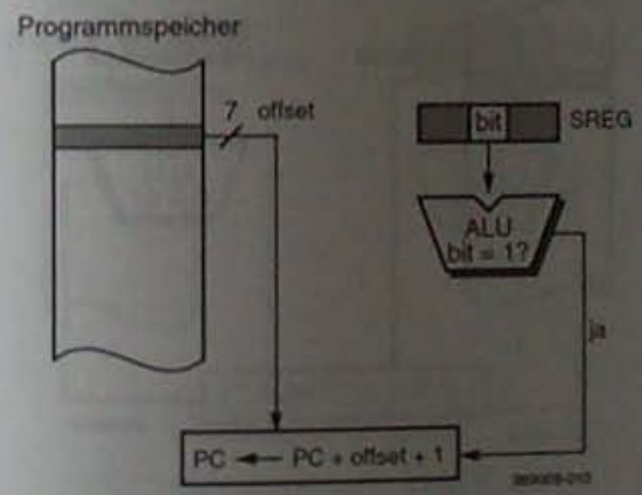
BRBS bit,offset

BRBS bit,offset — Springe relativ um den Wert offset falls bit im SREG gesetzt.

Funktion: PC ← PC + 1 + offset, falls SREG<bit> = 1

Beschreibung: Ein relativer Sprung um den Wert offset wird ausgeführt, falls ein einzelnes Bit des STATUS-Register SREG an der Position bit gesetzt ist. Mit dem Wert bit läßt sich jedes Bit im STATUS-Register adressieren. Der Wert offset wird als Zweierkomplement interpretiert, so daß relative Sprünge im Bereich von −64 bis +63 relativ zum Programmzähler PC ausgeführt werden können.

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus (2 bei Sprung), 1 Taktimpuls (2 bei Sprung): 125 ns (250 ns bei Sprung) bei 8 MHz

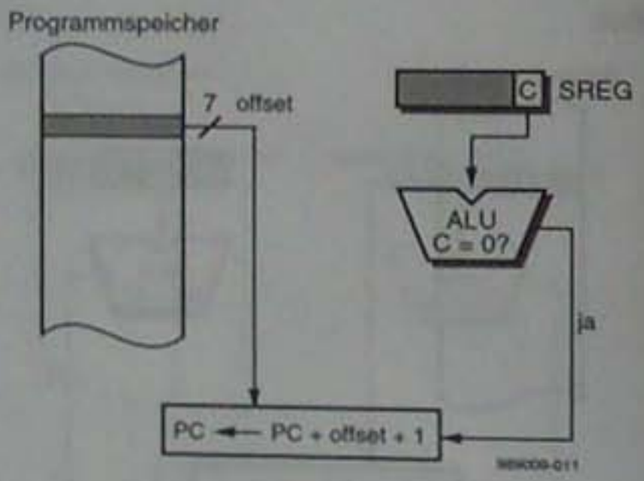
Flags:

I	T	H	S	V	N	Z	C

BRCC offset

BRCC offset	Springe relativ um den Wert offset falls Carry gelöscht.
Funktion:	PC ← PC + 1 + offset, falls (C) = 0
Beschreibung:	Ein relativer Sprung um den Wert offset wird ausgeführt, falls das Carry-Flag im STATUS-Register SREG gelöscht ist. Der Wert offset wird als Zweierkomplement interpretiert, so daß relative Sprünge im Bereich von −64 bis +63 relativ zum Programmzähler PC ausgeführt werden können.

Datenfluß:

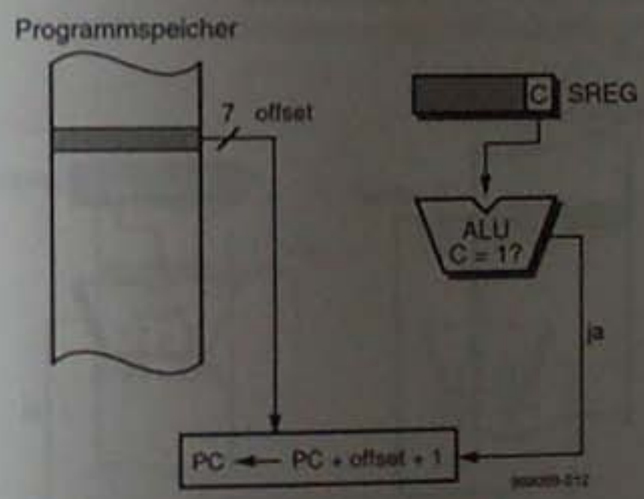


Befehlsablauf:	1 Wort, 1 Maschinenzyklus (2 bei Sprung), 1 Taktimpuls (2 bei Sprung): 125 ns (250 ns bei Sprung) bei 8 MHz
Flags:	I T H S V N Z C

BRCS offset

BRCS offset	Springe relativ um den Wert offset falls Carry gesetzt.
Funktion:	PC ← PC + 1 + offset, falls C = 1
Beschreibung:	Ein relativer Sprung um den Wert offset wird ausgeführt, falls das Carry-Flag im STATUS-Register SREG gesetzt ist. Der Wert offset wird als Zweierkomplement interpretiert, so daß relative Sprünge im Bereich von −64 bis +63 relativ zum Programmzähler PC ausgeführt werden können.

Datenfluß:

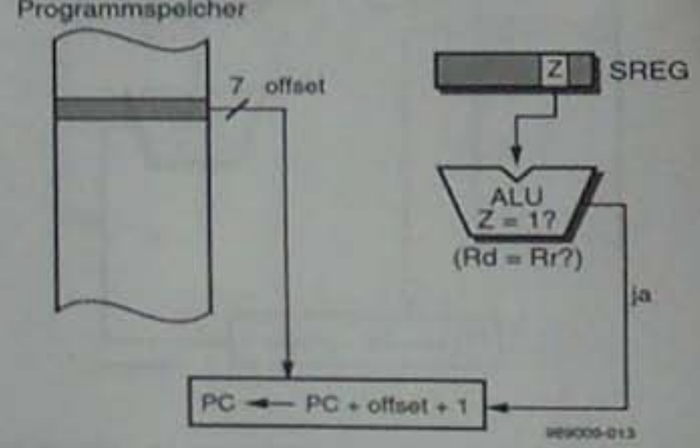


Befehlsablauf:	1 Wort, 1 Maschinenzyklus (2 bei Sprung), 1 Taktimpuls (2 bei Sprung): 125 ns (250 ns bei Sprung) bei 8 MHz
Flags:	I T H S V N Z C

BREQ offset

BREQ offset	Springe relativ um den Wert offset falls gleich.
Funktion:	PC ← PC + 1 + offset, falls Rd = Rr, (Z) = 1
Beschreibung:	Ein relativer Sprung um den Wert offset wird ausgeführt, falls das Zero-Flag im STATUS-Register SREG gesetzt ist. Steht dieser Befehl unmittelbar nach dem Befehl CP, CPI, SUB oder SUBI, dann wird der Sprung nur ausgeführt, falls die Werte in den Registern Rd und Rr gleich sind. Der Wert offset wird als Zweierkomplement interpretiert, so daß relative Sprünge im Bereich von −64 bis +63 relativ zum Programmzähler PC ausgeführt werden können.

Datenfluß:

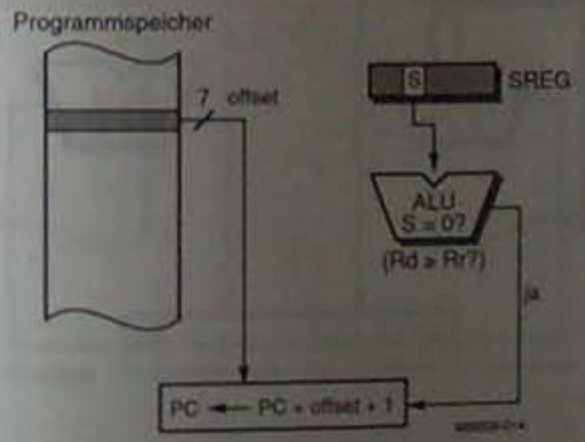


Befehlsablauf:	1 Wort, 1 Maschinenzyklus (2 bei Sprung), 1 Taktimpuls (2 bei Sprung): 125 ns (250 ns bei Sprung) bei 8 MHz
Flags:	I T H S V N Z C

BRGE offset

BRGE offset	Springe relativ um den Wert offset falls größer oder gleich.
Funktion:	PC ← PC + 1 + offset, falls Rd ≥ Rr, S = 0
Beschreibung:	Ein relativer Sprung um den Wert offset wird ausgeführt, falls das S-Flag im STATUS-Register SREG gelöscht ist. Steht dieser Befehl unmittelbar nach dem Befehl CP, CPI, SUB oder SUBI, dann wird der Sprung nur ausgeführt, falls der vorzeichenbehaftete Wert im Registern Rd größer oder gleich dem vorzeichenbehafteten Wert im Register Rr ist. Der Wert offset wird als Zweierkomplement interpretiert, so daß relative Sprünge im Bereich von −64 bis +63 relativ zum Programmzähler PC ausgeführt werden können.

Datenfluß:



Befehlsablauf:	1 Wort, 1 Maschinenzyklus (2 bei Sprung), 1 Taktimpuls (2 bei Sprung): 125 ns (250 ns bei Sprung) bei 8 MHz
Flags:	I T H S V N Z C

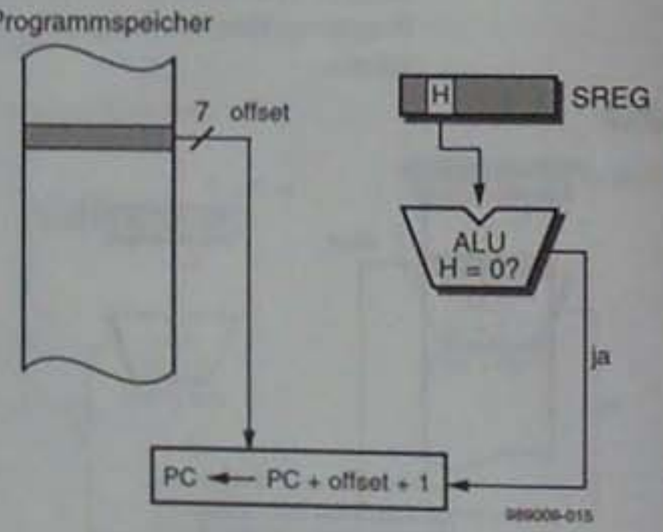
BRHC offset

BRHC offset — Springe relativ um den Wert offset falls Half-Carry gelöscht.

Funktion: PC ← PC + 1 + offset, falls H = 0

Beschreibung: Ein relativer Sprung um den Wert offset wird ausgeführt, falls das Half-Carry-Flag im STATUS-Register SREG gelöscht ist. Der Wert offset wird als Zweierkomplement interpretiert, so daß relative Sprünge im Bereich von –64 bis +63 relativ zum Programmzähler PC ausgeführt werden können.

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus (2 bei Sprung), 1 Taktimpuls (2 bei Sprung): 125 ns (250 ns bei Sprung) bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

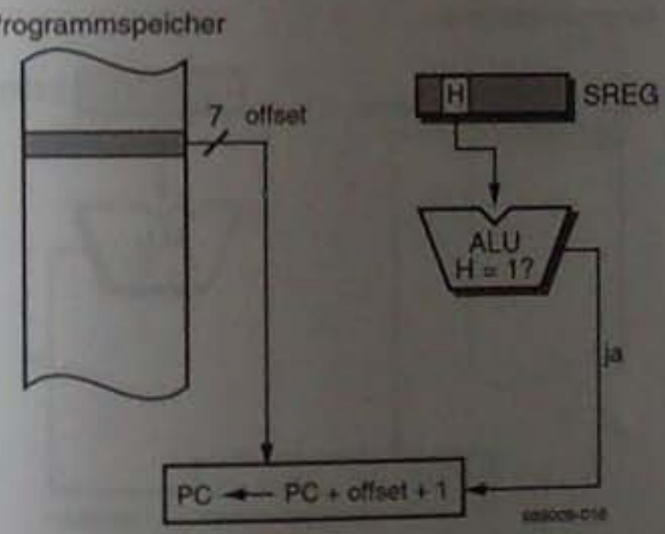
BRHS offset

BRHS offset — Springe relativ um den Wert offset falls Half-Carry gesetzt.

Funktion: PC ← PC + 1 + offset, falls H = 1

Beschreibung: Ein relativer Sprung um den Wert offset wird ausgeführt, falls das Half-Carry-Flag im STATUS-Register SREG gesetzt ist. Der Wert offset wird als Zweierkomplement interpretiert, so daß relative Sprünge im Bereich von –64 bis +63 relativ zum Programmzähler PC ausgeführt werden können.

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus (2 bei Sprung), 1 Taktimpuls (2 bei Sprung): 125 ns (250 ns bei Sprung) bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

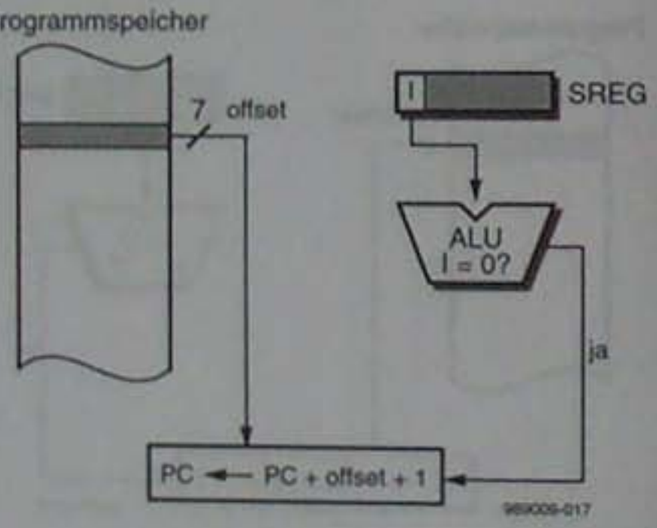
BRID offset

BRID offset — Springe relativ um den Wert offset falls Interrupt-Flag gelöscht.

Funktion: PC ← PC + 1 + offset, falls I = 0

Beschreibung: Ein relativer Sprung um den Wert offset wirdausgeführt, falls das Global Interrupt Flag im STATUS-Register SREG gelöscht ist. Der Wert offset wird als Zweierkomplement interpretiert, so daß relative Sprünge im Bereich von −64 bis +63 relativ zum Programmzähler PC ausgeführt werden können.

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus (2 bei Sprung).
1 Taktimpuls (2 bei Sprung):
125 ns (250 ns bei Sprung) bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

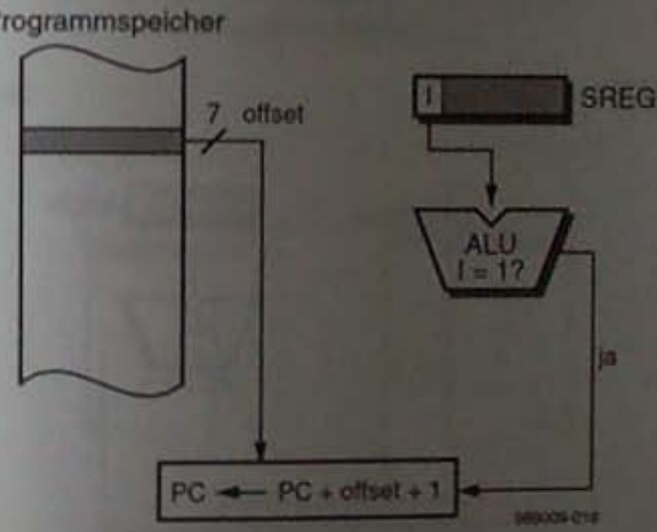
BRIE offset

BRIE offset — Springe relativ um den Wert offset falls Interrupt-Flag gesetzt.

Funktion: PC ← PC + 1 + offset, falls I = 1

Beschreibung: Ein relativer Sprung um den Wert offset wird ausgeführt, falls das Global Interrupt Flag im STATUS-Register SREG gesetzt ist. Der Wert offset wird als Zweierkomplement interpretiert, so daß relative Sprünge im Bereich von −64 bis +63 relativ zum Programmzähler PC ausgeführt werden können.

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus (2 bei Sprung).
1 Taktimpuls (2 bei Sprung):
125 ns (250 ns bei Sprung) bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

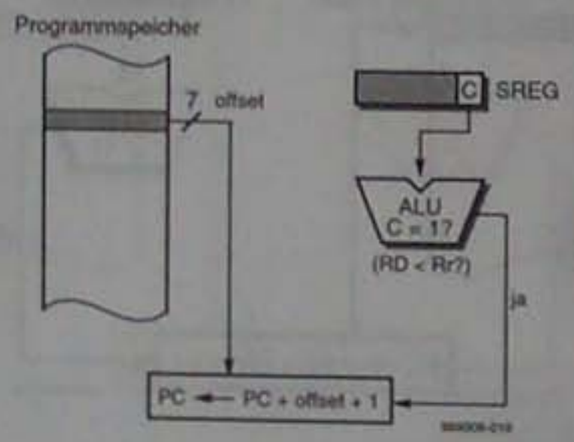
BRLO offset

BRLO offset — Springe relativ um den Wert offset falls niedriger.

Funktion: PC ← PC + 1 + offset, falls Rd < Rr, C = 1

Beschreibung: Ein relativer Sprung um den Wert offset wird ausgeführt, falls das Carry-Flag im STATUS-Register SREG gesetzt ist. Steht dieser Befehl unmittelbar nach dem Befehl CP, CPI, SUB oder SUBI, dann wird der Sprung nur ausgeführt, falls der vorzeichenlose Wert im Register Rd niedriger ist als der vorzeichenlose Wert im Register Rr ist. Der Wert offset wird als Zweierkomplement interpretiert, so daß relative Sprünge im Bereich von −64 bis +63 relativ zum Programmzähler PC ausgeführt werden können.

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus (2 bei Sprung), 1 Taktimpuls (2 bei Sprung): 125 ns (250 ns bei Sprung) bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

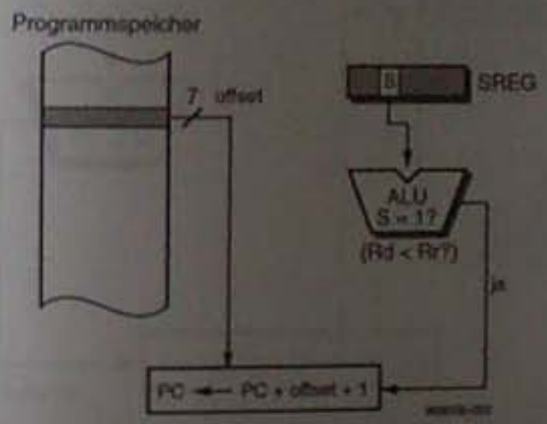
BRLT offset

BRLT offset — Springe relativ um den Wert offset falls kleiner.

Funktion: PC ← PC + 1 + offset, falls Rd < Rr, S = 1

Beschreibung: Ein relativer Sprung um den Wert offset wird ausgeführt, falls das Signed-Flag im STATUS-Register SREG gesetzt ist. Steht dieser Befehl unmittelbar nach dem Befehl CP, CPI, SUB oder SUBI, dann wird der Sprung nur ausgeführt, falls der vorzeichenbehaftete Wert im Register Rd kleiner ist als der vorzeichenbehaftete Wert im Register Rr ist. Der Wert offset wird als Zweierkomplement interpretiert, so daß relative Sprünge im Bereich von −64 bis +63 relativ zum Programmzähler PC ausgeführt werden können.

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus (2 bei Sprung), 1 Taktimpuls (2 bei Sprung): 125 ns (250 ns bei Sprung) bei 8 MHz

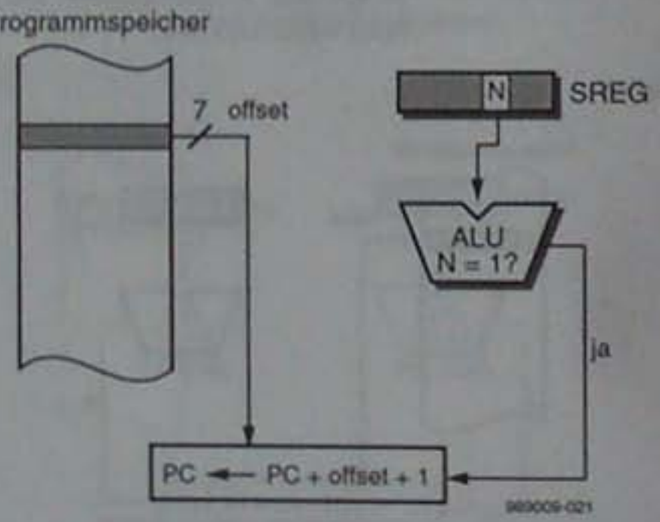
Flags:

I	T	H	S	V	N	Z	C

BRMI offset

BRMI offset	Springe relativ um den Wert offset falls negativ.
Funktion:	PC ← PC + 1 + offset, falls N = 1
Beschreibung:	Ein relativer Sprung um den Wert offset wird ausgeführt, falls das Negative-Flag im STATUS-Register SREG gesetzt ist. Der Wert offset wird als Zweierkomplement interpretiert, so daß relative Sprünge im Bereich von –64 bis +63 relativ zum Programmzähler PC ausgeführt werden können.

Datenfluß:

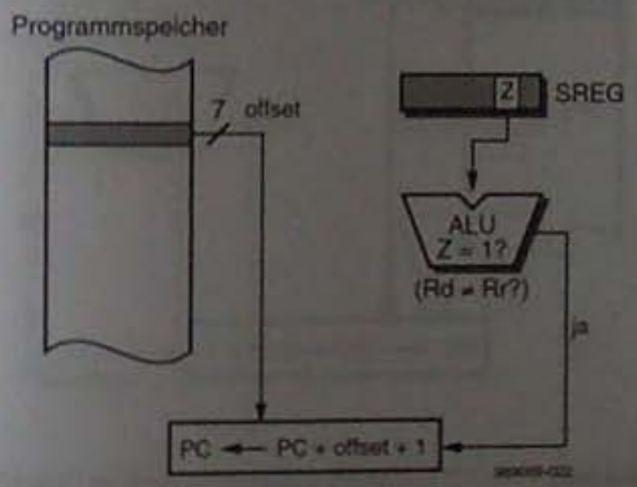


Befehlsablauf:	1 Wort, 1 Maschinenzyklus (2 bei Sprung), 1 Taktimpuls (2 bei Sprung): 125 ns (250 ns bei Sprung) bei 8 MHz
Flags:	I T H S V N Z C

BRNE offset

BRNE offset	Springe relativ um den Wert offset falls ungleich.
Funktion:	PC ← PC + 1 + offset, falls Rd ≠ Rr, Z = 0
Beschreibung:	Ein relativer Sprung um den Wert offset wird ausgeführt, falls das Zero-Flag im STATUS-Register SREG gelöscht ist. Steht dieser Befehl unmittelbar nach dem Befehl CP, CPI, SUB oder SUBI, dann wird der Sprung nur ausgeführt, falls die Werte in den Registern Rd und Rr ungleich sind. Der Wert offset wird als Zweierkomplement interpretiert, so daß relative Sprünge im Bereich von –64 bis +63 relativ zum Programmzähler PC ausgeführt werden können.

Datenfluß:



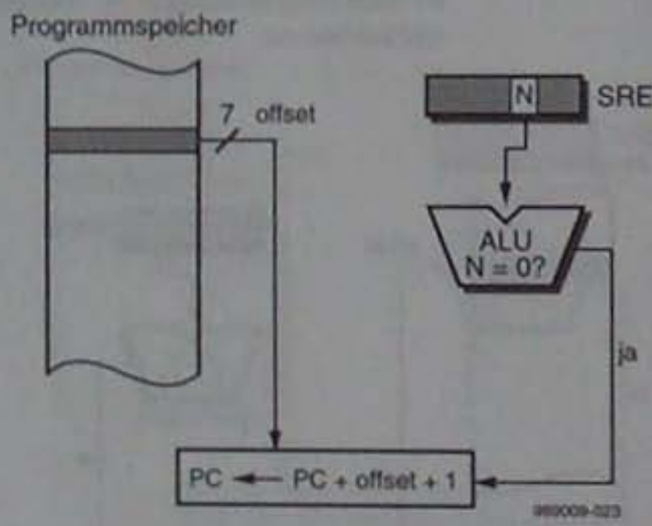
Befehlsablauf:	1 Wort, 1 Maschinenzyklus (2 bei Sprung), 1 Taktimpuls (2 bei Sprung): 125 ns (250 ns bei Sprung) bei 8 MHz
Flags:	I T H S V N Z C

BRNE offset

BRPL offset

BRPL offset	Springe relativ um den Wert offset falls positiv.
Funktion:	PC ← PC + 1 + offset, falls N = 0
Beschreibung:	Ein relativer Sprung um den Wert offset wird ausgeführt, falls das Negative-Flag im STATUS-Register SREG gelöscht ist. Der Wert offset wird als Zweierkomplement interpretiert, so daß relative Sprünge im Bereich von –64 bis +63 relativ zum Programmzähler PC ausgeführt werden können.

Datenfluß:

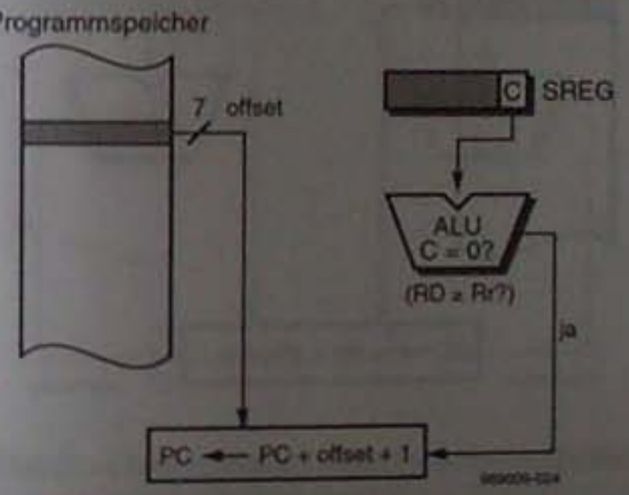


Befehlsablauf:	1 Wort, 1 Maschinenzyklus (2 bei Sprung), 1 Taktimpuls (2 bei Sprung): 125 ns (250 ns bei Sprung) bei 8 MHz
Flags:	I T H S V N Z C

BRSH offset

BRSH offset	Springe relativ um den Wert offset falls gleich oder höher.
Funktion:	PC ← PC + 1 + offset, falls Rd >= Rr, C = 0
Beschreibung:	Ein relativer Sprung um den Wert offset wirdausgeführt, falls das Carry-Flag im STATUS-Register SREG gelöscht ist. Steht dieser Befehl unmittelbar nach dem Befehl CP, CPI, SUB oder SUBI, dann wird der Sprung nur ausgeführt, falls der vorzeichenlose Wert im Registern Rd gleich oder höher dem vorzeichenlosen Wert im Register Rr ist. Der Wert offset wird als Zweierkomplement interpretiert, so daß relative Sprünge im Bereich von –64 bis +63 relativ zum Programmzähler PC ausgeführt werden können.

Datenfluß:



Befehlsablauf:	1 Wort, 1 Maschinenzyklus (2 bei Sprung), 1 Taktimpuls (2 bei Sprung): 125 ns (250 ns bei Sprung) bei 8 MHz
Flags:	I T H S V N Z C

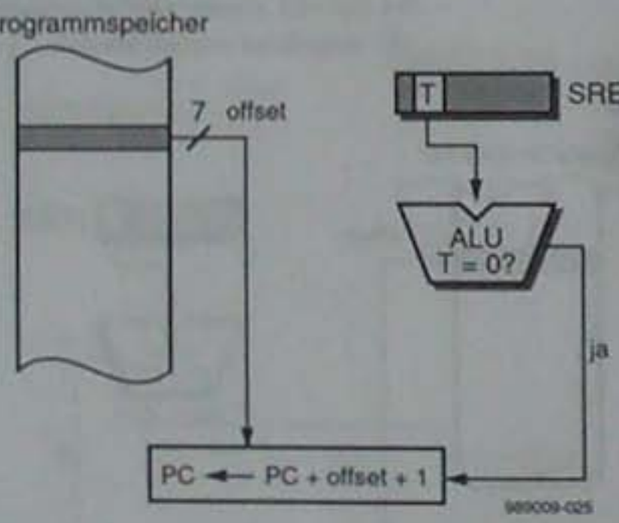
BRTC offset

BRTC offset — Springe relativ um den Wert offset falls T-Flag gelöscht.

Funktion: PC ← PC + 1 + offset, falls T = 0

Beschreibung: Ein relativer Sprung um den Wert offset wird ausgeführt, falls das Transfer-Bit im STATUS-Register SREG gelöscht ist. Der Wert offset wird als Zweierkomplement interpretiert, so daß relative Sprünge im Bereich von −64 bis +63 relativ zum Programmzähler PC ausgeführt werden können.

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus (2 bei Sprung), 1 Taktimpuls (2 bei Sprung): 125 ns (250 ns bei Sprung) bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

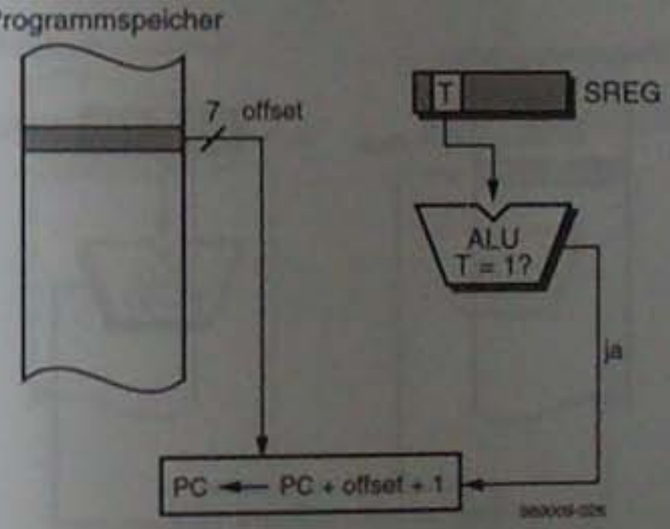
BRTS offset

BRTS offset — Springe relativ um den Wert offset falls T-Flag gesetzt.

Funktion: PC ← PC + 1 + offset, falls T = 1

Beschreibung: Ein relativer Sprung um den Wert offset wird ausgeführt, falls das Transfer-Bit im STATUS-Register SREG gesetzt ist. Der Wert offset wird als Zweierkomplement interpretiert, so daß relative Sprünge im Bereich von −64 bis +63 relativ zum Programmzähler PC ausgeführt werden können.

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus (2 bei Sprung), 1 Taktimpuls (2 bei Sprung): 125 ns (250 ns bei Sprung) bei 8 MHz

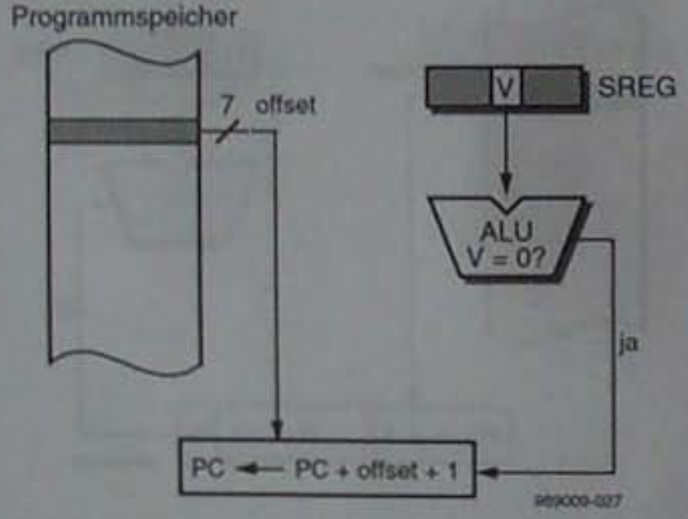
Flags:

I	T	H	S	V	N	Z	C

BRVC offset

BRVC offset	Springe relativ um den Wert offset falls V-Flag gelöscht.
Funktion:	PC ← PC + 1 + offset, falls V = 0
Beschreibung:	Ein relativer Sprung um den Wert offset wird ausgeführt, falls das Zweierkomplement-Überlauf Flag im STATUS-Register SREG gelöscht ist. Der Wert offset wird als Zweierkomplement interpretiert, so daß relative Sprünge im Bereich von −64 bis +63 relativ zum Programmzähler PC ausgeführt werden können.

Datenfluß:

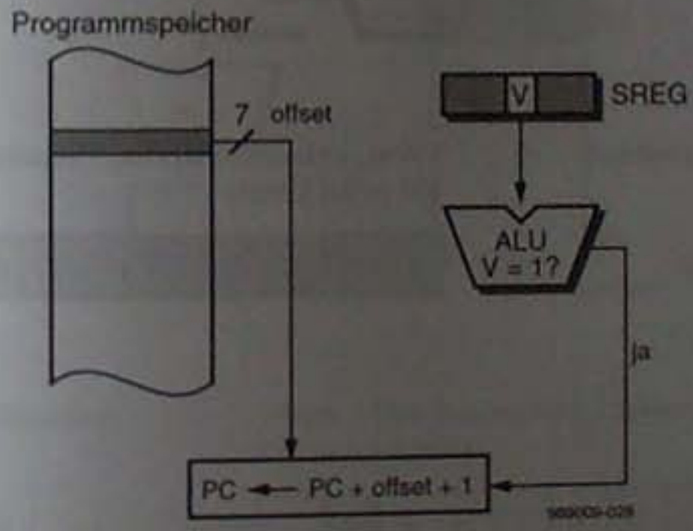


Befehlsablauf:	1 Wort, 1 Maschinenzyklus (2 bei Sprung). 1 Taktimpuls (2 bei Sprung): 125 ns (250 ns bei Sprung) bei 8 MHz
Flags:	I T H S V N Z C

BRVS offset

BRVS offset	Springe relativ um den Wert offset falls V-Flag gesetzt.
Funktion:	PC ← PC + 1 + offset, falls V = 1
Beschreibung:	Ein relativer Sprung um den Wert offset wird ausgeführt, falls das Zweierkomplement-Überlauf Flag im STATUS-Register SREG gesetzt ist. Der Wert offset wird als Zweierkomplement interpretiert, so daß relative Sprünge im Bereich von −64 bis +63 relativ zum Programmzähler PC ausgeführt werden können.

Datenfluß:



Befehlsablauf:	1 Wort, 1 Maschinenzyklus (2 bei Sprung). 1 Taktimpuls (2 bei Sprung): 62,5 ns (125 ns bei Sprung) bei 16 MHz
Flags:	I T H S V N Z C

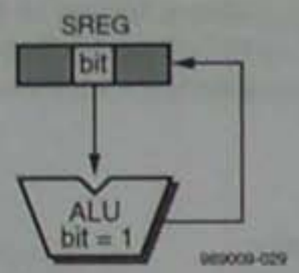
BSET bit

BSET bit — Setze bit im SREG.

Funktion: SREG<bit> ← 1

Beschreibung: Das Flag an der Position bit wird im STATUS-Register SREG gesetzt. Mit dem Wert bit läßt sich jedes Flag im STATUS-Register adressieren.

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus, 1 Taktimpuls: 125 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

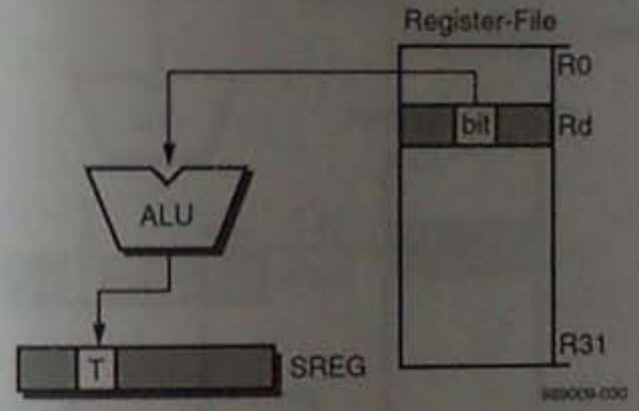
BST Rd,bit

BST Rd,bit — Speichere bit von Rd im T-Bit.

Funktion: T ← Rd<bit>

Beschreibung: Ein Bit des Registers Rd, das an der Position bit steht, wird in das Transfer-Bit im STATUS-Register SREG gespeichert. Mit dem Wert bit läßt sich jedes Bit im Register Rd adressieren. Der Befehl ist für alle Register R0 bis R31 im Register File zulässig.

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus, 1 Taktimpuls: 125 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

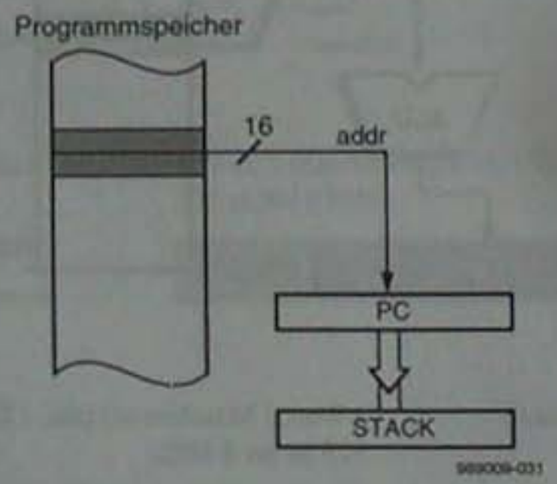
CALL addr

CALL addr — Aufruf eines Unterprogramms an der Adresse addr.

Funktion: PC ← addr

Beschreibung: Ein Unterprogramm an der Adresse addr wird aufgerufen. Der um zwei erhöhte Programmzähler PC wird auf den Stack geschoben. Der Programmzähler wird mit der Adresse addr im Operanden geladen. Das Unterprogramm kann sich an einer beliebigen Adresse im Programmspeicher befinden. (Nur im ATmega103/603 implementiert.)

Datenfluß:



Befehlsablauf: 2 Worte, 4 Maschinenzyklen, 4 Taktimpulse: 500 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

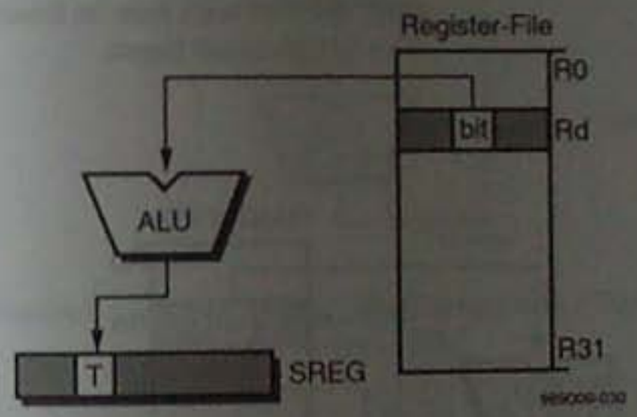
CBI Port,bit

CBI Port,bit — Lösche ein Bit im Port-Register

Funktion: Port<bit> ← 0

Beschreibung: Löscht das angegebene Bit in einem Port-Register. Der Wert bit kann im Bereich von 0 bis 7 liegen. Der Wert Port kann im Bereich von 0 bis 31 liegen.

Datenfluß:



Befehlsablauf: 1 Wort, 2 Maschinenzyklen, 2 Taktimpulse: 250 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

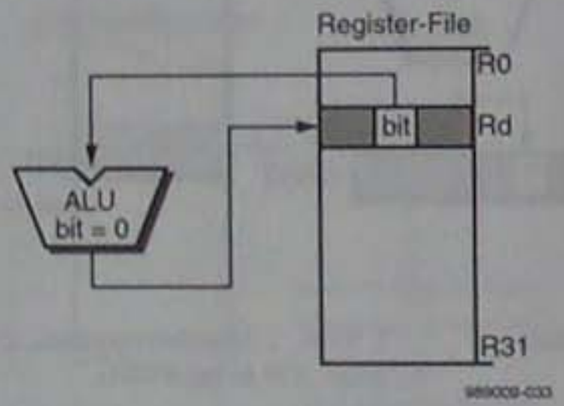
CBR Rd,mask

CBR Rd,mask — Lösche Bit im Regsiter Rd.

Funktion: Rd ← Rd .AND. (SFF - mask)

Beschreibung: Löscht ein Bit im Register Rd. Das zu löschende Bit wird mit dem Wert mask maskiert. Soll z. B. das Bit 3 im gelöscht werden, so muß mask den Wert 00001000b (binär) bzw. $08 (hex) annehmen. Der Befehl ist für die Register R16 bis R31 in der oberen Hälfte des Register-Files zulässig. Der Wert mask kann im Bereich von 0 bis 255 (dezimal) liegen.

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus, 1 Taktimpuls: 125 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

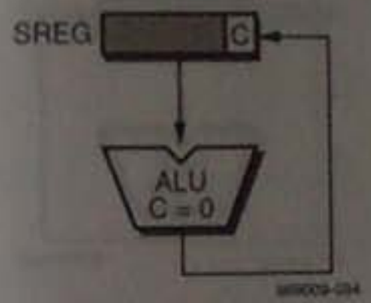
CLC

CLC — Lösche Carry.

Funktion: C ← 0

Beschreibung: Das Carry-Flag im STATUS-Register SREG wird gelöscht.

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus, 1 Taktimpuls: 125 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

CLH

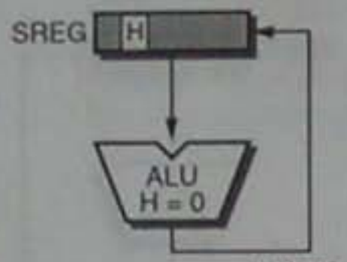
CLH

Lösche Half-Carry.

Funktion: $H \leftarrow 0$

Beschreibung: Das Half-Carry-Flag im STATUS-Register SREG wird gelöscht.

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus, 1 Taktimpuls: 125 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

CLI

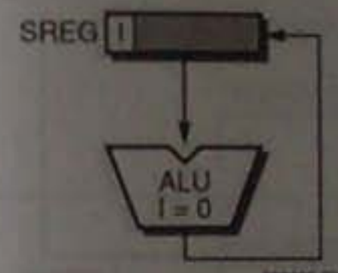
Lösche Interrupt-Flag.

CLI

Funktion: $I \leftarrow 0$

Beschreibung: Das Global-Interrupt-Flag im STATUS-Register SREG wird gelöscht.

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus, 1 Taktimpuls: 125 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

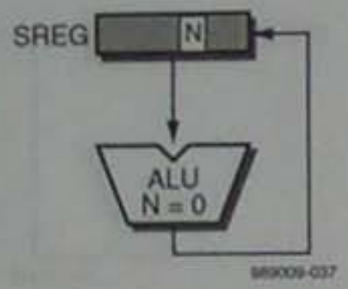
CLN

CLN — Lösche Negative-Flag.

Funktion: N ← 0

Beschreibung: Das Negative-Flag im STATUS-Register SREG wird gelöscht.

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus, 1 Taktimpuls: 125 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

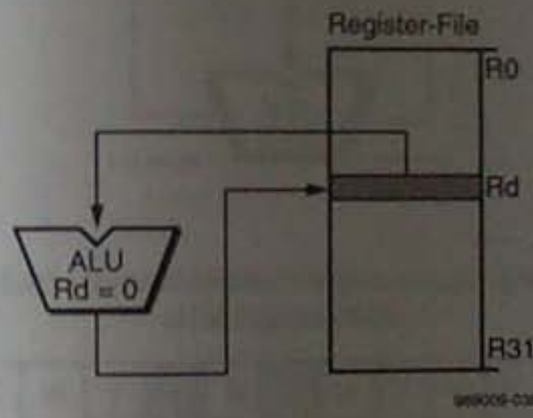
CLR Rd

CLR Rd — Lösche Register Rd.

Funktion: Rd ← Rd .XOR. Rd

Beschreibung: Das Register Rd wird gelöscht, indem es mit sich selber XOR-verknüpft wird. Der Befehl ist für alle Register R0 bis R31 im Register-File zulässig.

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus, 1 Taktimpuls: 125 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

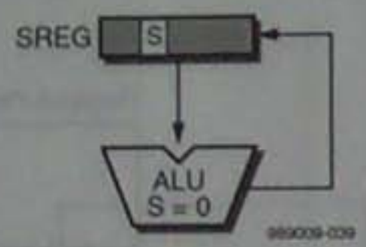
CLS

CLS — Lösche Signed-Flag.

Funktion: S ← 0

Beschreibung: Das Signed-Flag im STATUS-Register SREG wird gelöscht.

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus, 1 Taktimpuls: 125 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

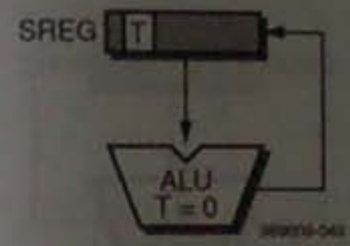
CLT

CLT — Lösche T-Flag.

Funktion: T ← 0

Beschreibung: Das Transfer-Bit im STATUS-Register SREG wird gelöscht.

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus, 1 Taktimpuls: 125 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

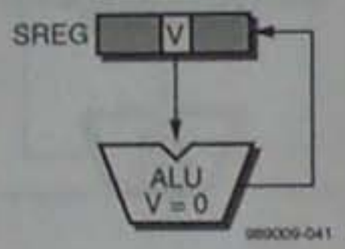
CLV

CLV — Lösche V-Flag.

Funktion: V ← 0

Beschreibung: Der Zweierkomplement-Überlauf Indikator im STATUS-Register SREG wird gelöscht.

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus, 1 Taktimpuls: 125 ns bei 8 MHz

Flags:

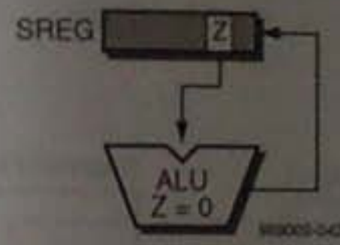
I	T	H	S	V	N	Z	C

CLZ — Lösche Zero-Flag.

Funktion: Z ← 0

Beschreibung: Das Zero-Flag im STATUS-Register SREG wird gelöscht.

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus, 1 Taktimpuls: 125 ns bei 8 MHz

Flags:

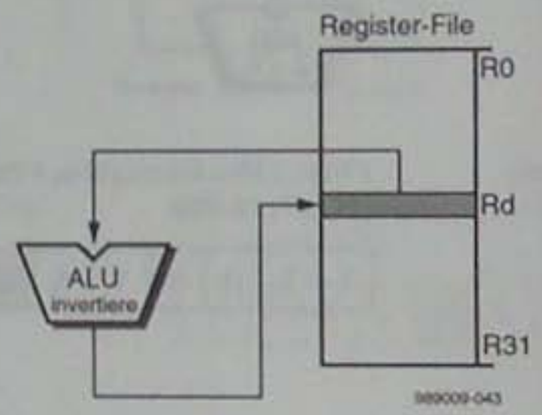
I	T	H	S	V	N	Z	C

CLZ

COM Rd

COM Rd	Bilde das Einerkomplement des Registers Rd.
Funktion:	Rd ← $FF - Rd
Beschreibung:	Das Einerkomplement aus dem Inhalt des Registers Rd wird gebildet. Dazu wird vom konstanten Wert $FF (hex) der Inhalt des Registers Rd subtrahiert. Der Befehl ist für alle Register R0 bis R31 im Register-File zulässig.

Datenfluß:

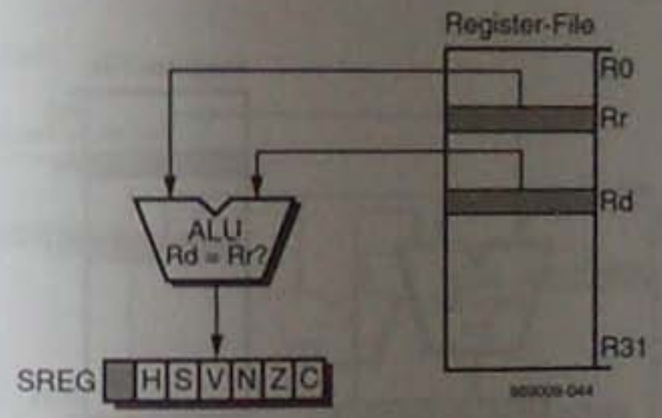


Befehlsablauf:	1 Wort, 1 Maschinenzyklus, 1 Taktimpuls: 125 ns bei 8 MHz
Flags:	I T H S V N Z C

CP Rd,Rr

CP Rd,Rr	Vergleiche Register Rd mit Rr.
Funktion:	Rd - Rr
Beschreibung:	Der Inhalt der Register Rd und Rr werden miteinander verglichen. Der Inhalt beider Register bleibt erhalten. Nach diesem Befehl können alle bedingte Verzweigungsbefehle folgen. Der Befehl ist für alle Register R0 bis R31 im Register-File zulässig.

Datenfluß:



Befehlsablauf:	1 Wort, 1 Maschinenzyklus, 1 Taktimpuls: 125 ns bei 8 MHz
Flags:	I T H S V N Z C

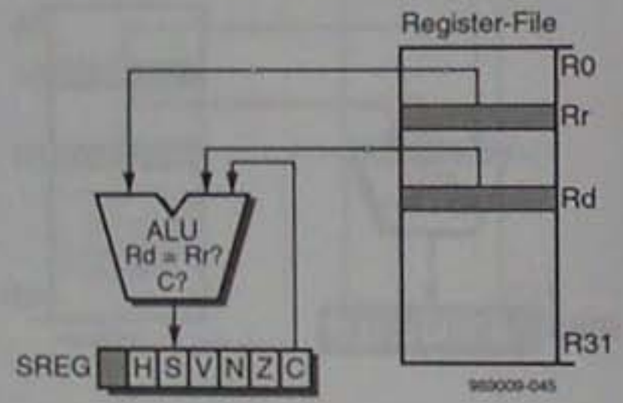
CPC Rd,Rr

CPC Rd,Rr — Vergleiche Register Rd mit Rr und Carry.

Funktion: Rd - Rr - C

Beschreibung: Der Inhalt der Register Rd und Rr werden miteinander verglichen. Dabei wird auch das Carry-Flag einbezogen. Der Inhalt beider Register bleibt erhalten. Nach diesem Befehl können alle bedingte Verzweigungsbefehle folgen. Der Befehl ist für alle Register R0 bis R31 im Register-File zulässig.

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus, 1 Taktimpuls: 125 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

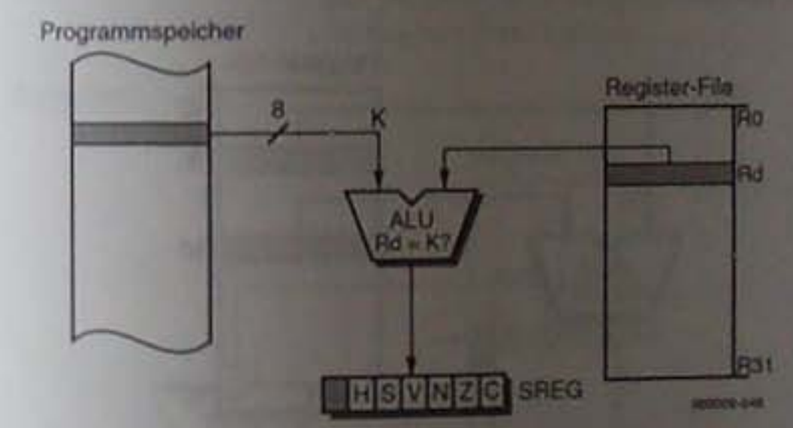
CPI Rd,K

CPI Rd,K — Vergleiche Register Rd mit unmittelbaren Wert K.

Funktion: Rd - K

Beschreibung: Der Inhalt der Register Rd und der unmittelbare Wert K werden miteinander verglichen. Der Inhalt des Registers bleibt erhalten. Nach diesem Befehl können alle bedingte Verzweigungsbefehle folgen. Der Befehl ist für die Register R16 bis R31 in der oberen Hälfte des Register-Files zulässig. Der unmittelbare Wert K kann im Bereich von 0 bis 255 (dezimal) liegen.

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus, 1 Taktimpuls: 125 ns bei 8 MHz

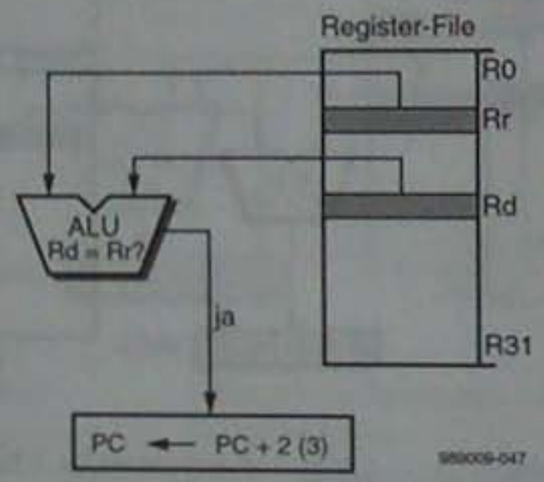
Flags:

I	T	H	S	V	N	Z	C

CPSE Rd,Rr

CPSE Rd,Rr	Vergleiche Register Rd mit Rr und springe falls gleich.
Funktion:	PC ← PC + 2 (oder 3), falls Rd = Rr
Beschreibung:	Der Inhalt der Register Rd und Rr werden miteinander verglichen. Sind die Inhalte beider Register gleich, wird der nächste Befehl übersprungen. Handelt es sich beim nächsten Befehl um ein Einwort-Befehl, wird der Programmzähler PC um zwei erhöht. Bei einem Zweiwort-Befehl wird der Programmzähler um drei erhöht. Der Befehl ist für alle Register R0 bis R31 im Register-File zulässig.

Datenfluß:

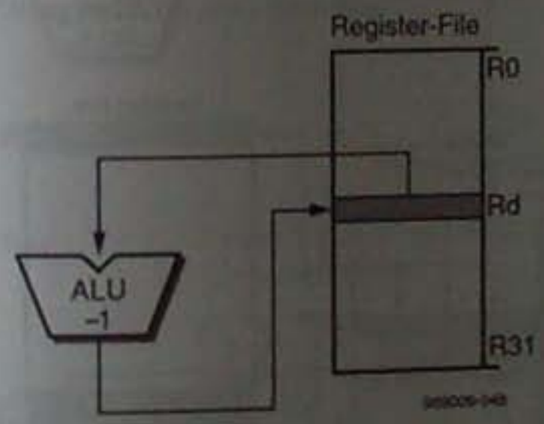


Befehlsablauf:	1 Wort, 1 Maschinenzyklus, 1 Taktimpuls: 125 ns bei 8 MHz
Flags:	I T H S V N Z C

DEC Rd

DEC Rd	Dekrementiere Register Rd.
Funktion:	Rd ← Rd - 1
Beschreibung:	Der Inhalt des Registers Rd wird um eins dekrementiert. Das Ergebnis wird in das Register Rd zurückgespeichert. Das Carry-Flag wird nicht beeinflußt, so daß der Befehl in Schleifen für aritmetische Berechnungen benutzt werden kann. Der DEC-Befehl arbeitet bei vorzeichenlosen Werten nur mit den bedingten Verzweigungsbefehlen BREQ und BRNE konsistent zusammen. Werden hingegen Werte in Zweierkomplementdarstellung verwendet, stehen alle bedingte Verzweigungsbefehle für vorzeichenbehaftete Werte zur Verfügung. Der Befehl ist für alle Register R0 bis R31 im Register-Files zulässig.

Datenfluß:

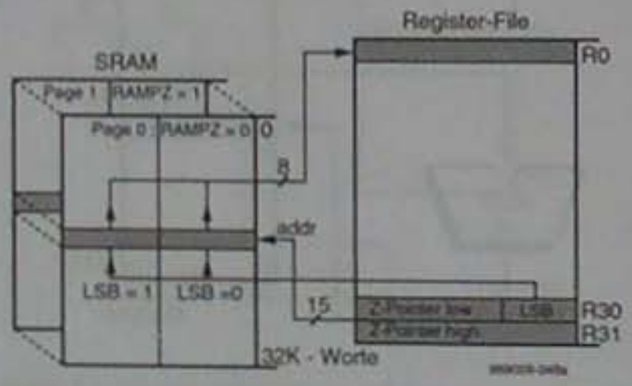


Befehlsablauf:	1 Wort, 1 Maschinenzyklus, 1 Taktimpuls: 125 ns bei 8 MHz
Flags:	I T H S V N Z C

ELPM

ELPM	Lade Register R0 mit einem Byte aus dem Programmspeicher, auf das der Z-Pointer und RAMPZ zeigen.
Funktion:	R0 ← (Z+ RAMPZ)
Beschreibung:	Lädt Register R0 indirekt mit einem Byte aus dem Programmspeicher, das über den Inhalt des Z-Pointers (R31:R30) und dem RAMPZ-Register adressiert wird. Da der Programmspeicher 16-Bit breit ist, wird über das LSB des Z-Pointers angegeben, ob das obere oder untere Byte des Programmspeichers in das Register R0 geladen werden soll. Ist das LSB 0, so wird das untere Byte geladen. Hingegen wird bei 1 das obere Byte geladen. Der Befehl ist nur für das Register R0 aus dem Register-File zulässig. Mit RAMPZ = 0 wird die untere 64KByte-Page, mit RAMPZ = 1 die obere 64KByte-Page des ATmega103 adressiert. (Nur im ATmega103)

Datenfluß:



Befehlsablauf: 1 Wort, 3 Maschinenzyklen, 3 Taktimpuls: 375 ns bei 8 MHz

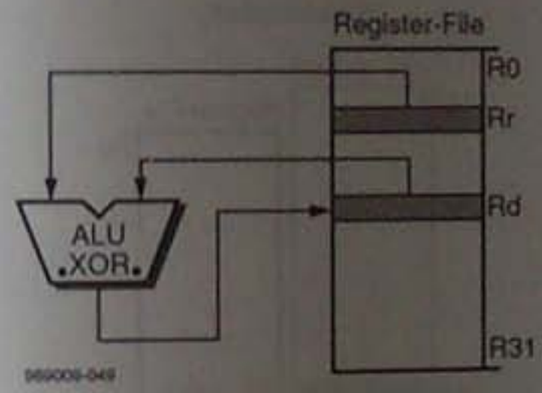
Flags:

I	T	H	S	V	N	Z	C

EOR Rd,Rr

EOR Rd,Rr	Register Rr und Rd XOR-verknüpfen.
Funktion:	Rd ← Rd .XOR. Rr
Beschreibung:	Der Inhalt des Registers Rr wird mit dem Inhalt des Registers Rd logisch XOR-verknüpft. Das Ergebnis der Verknüpfung steht im Register Rd. Der Inhalt des Registers Rr bleibt unverändert. Der Befehl ist für alle Register R0 bis R31 im Register File zulässig.

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus, 1 Taktimpuls: 125 ns bei 8 MHz

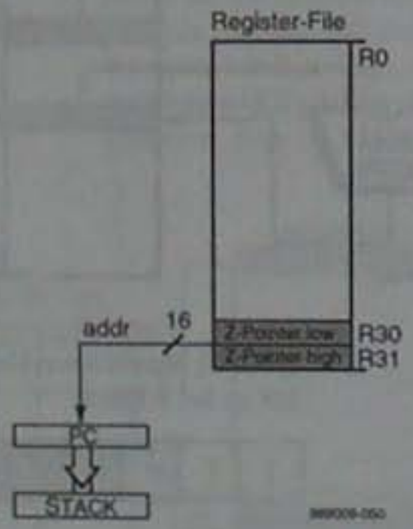
Flags:

I	T	H	S	V	N	Z	C

ICALL

ICALL	Indirekter Aufruf eines Unterprogramms.
Funktion:	PC ← R31:R30
Beschreibung:	Ein Unterprogramm an der Adresse, auf die der Inhalt des Pointers Z (R31:R30) zeigt, wird aufgerufen. Der um eins erhöhte Pro-grammzähler PC wird auf den Stack geschoben. Der Programmzähler wird mit der Adresse geladen, die im Pointer Z (R31:R30) steht. Das Unterprogramm kann sich an einer beliebigen Adresse innerhalb der 64K Worte im Programmspeicher befinden. (Nicht im AT90S1200 implementiert.)

Datenfluß:



Befehlsablauf:	1 Wort, 3 Maschinenzyklen, 3 Taktimpulse: 375 ns bei 8 MHz

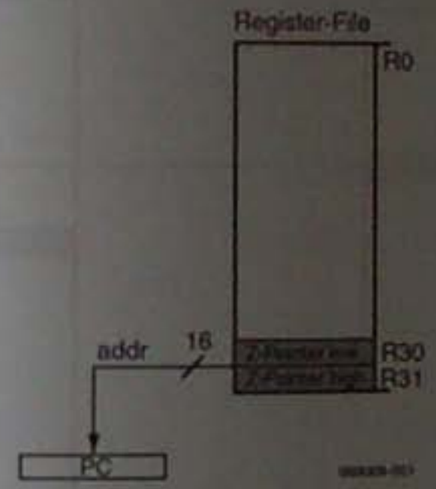
Flags:

I	T	H	S	V	N	Z	C

IJMP

IJMP	Indirekter Sprung.
Funktion:	PC ← R31:R30
Beschreibung:	Ein Sprung zur Adresse, auf die der Inhalt des Z-Pointers (R31:R30) zeigt, wird ausgeführt. Der Programmzähler wird mit der Adresse geladen, die im Z-Pointer (R31:R30) steht. Dieser Befehl kann an eine beliebigen Adresse innerhalb der 64K Worte im Programmspeicher verzweigen. (Nicht im AT90S1200 implementiert.)

Datenfluß:



Befehlsablauf:	1 Wort, 2 Maschinenzyklen, 2 Taktimpulse: 250 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

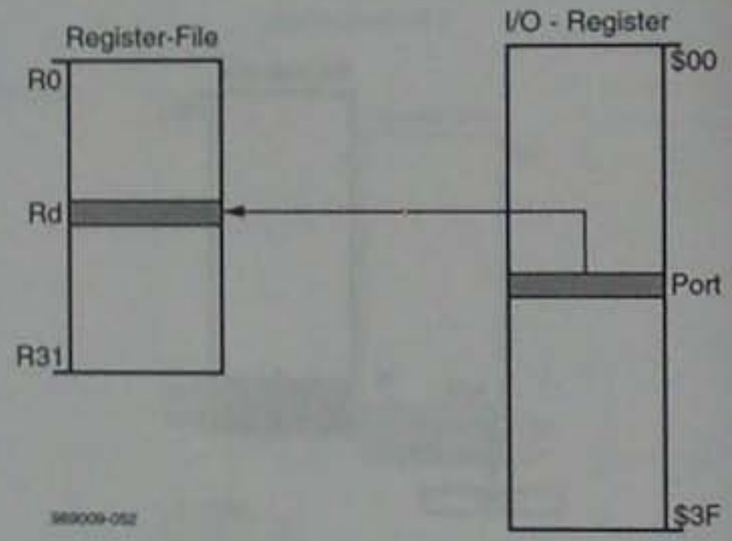
IN Rd,Port

IN Rd,Port — Lade Register Rd mit Port.

Funktion: Rd ← Port

Beschreibung: Lädt in das Register Rd den Inhalt des Registers, welches sich im I/O-Adreßraum befindet und durch den Wert Port adressiert wird. Der Befehl ist für alle Register R0 bis R31 im RegisterFile zulässig. Der Wert Port kann im Bereich von 0 bis 63 (dezimal) liegen.

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus, 1 Taktimpuls: 125 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

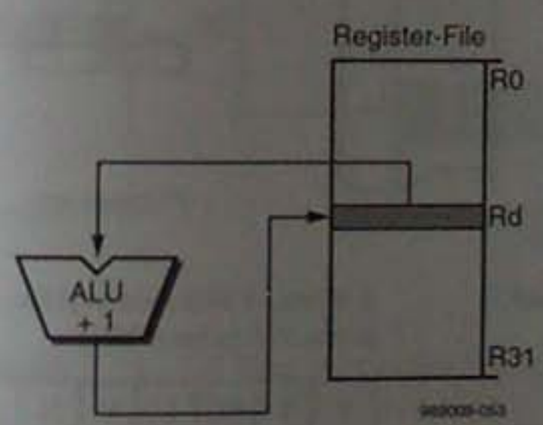
INC Rd

INC Rd — Inkrementiere Register Rd.

Funktion: Rd ← Rd + 1

Beschreibung: Der Inhalt des Registers Rd wird um eins inkrementiert. Das Ergebnis wird in das Register Rd zurückgespeichert. Das Carry-Flag wird nicht beeinflußt, so daß der Befehl in Schleifen für aritmetische Berechnungen benutzt werden kann. Der INC-Befehl arbeitet bei vorzeichenlosen Werten nur mit den bedingten Verzweigungsbefehlen BREQ und BRNE konsistent zusammen. Werden hingegen Werte in Zweierkomplementdarstellung verwendet, stehen alle bedingte Verzweigungsbefehle für vorzeichenbehaftete Werte zur Verfügung. Der Befehl ist für alle Register R0 bis R31 im Register-File zulässig.

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus, 1 Taktimpuls: 125 ns bei 8 MHz

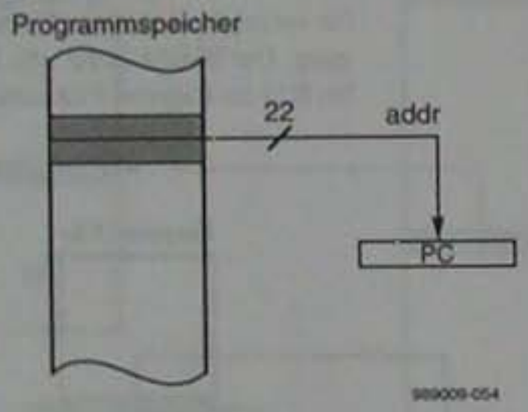
Flags:

I	T	H	S	V	N	Z	C

JMP addr

JMP addr	Sprung zur Adresse addr.
Funktion:	PC ← addr
Beschreibung:	Ein Sprung zur Adresse addr wird ausgeführt. Der Programmzähler wird mit der Adresse addr (22 Bit lang) geladen. Dieser Befehl kann an eine beliebigen Adresse innerhalb der 4M Worte im Programmspeicher verzweigen (vorbereitet für zukünftige Erweiterungen). (Nur im ATmega103/603 implementiert.)

Datenfluß:

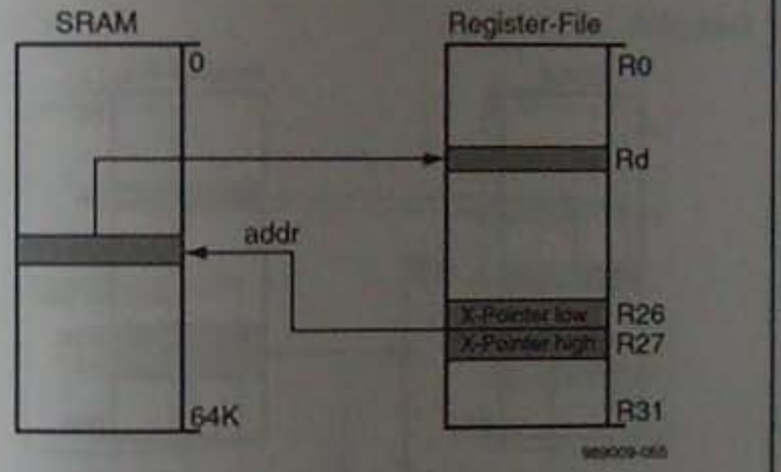


Befehlsablauf:	2 Worte, 3 Maschinenzyklen, 3 Taktimpulse: 375 ns bei 8 MHz
Flags:	I T H S V N Z C

LD Rd,X

LD Rd,X	Lade Register Rd indirekt über X-Pointer.
Funktion:	Rd ← (X)
Beschreibung:	Lädt in das Register Rd ein Byte aus dem SRAM, welches durch den Wert des X-Pointer (R27:R26) adressiert wird. Der Befehl ist für alle Register R0 bis R31 im Register-File zulässig. Der Zugriff auf das SRAM ist auf die selektierte 64KByte-Page beschränkt. (Nicht im AT90S1200 implementiert.)

Datenfluß:

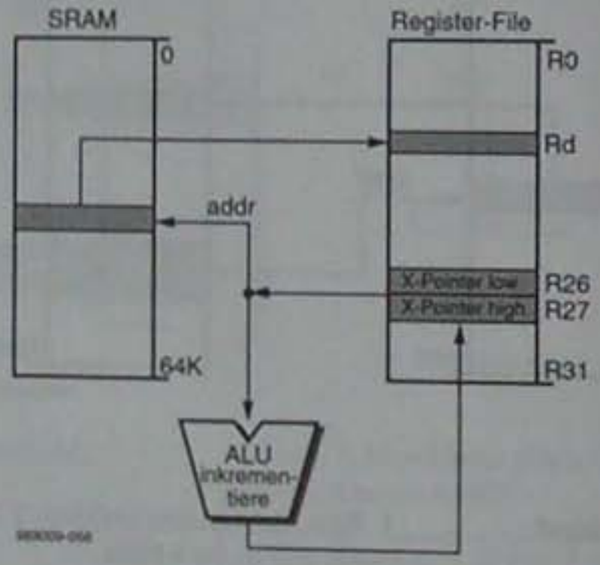


Befehlsablauf:	1 Wort, 2 Maschinenzyklen, 2 Taktimpulse: 250 ns bei 8 MHz
Flags:	I T H S V N Z C

LD Rd,X+

LD Rd,X+	Lade Register Rd indirekt über X-Pointer und inkrementiere Pointer nachher.
Funktion:	Rd ← (X); anschließend X = X + 1
Beschreibung:	Lädt in das Register Rd ein Byte aus dem SRAM, welches durch den Wert des X-Pointer (R27:R26) adressiert wird. Anschließend wird der X-Pointer um eins inkrementiert. Der Befehl ist für alle Register R0 bis R31 im Register-File zulässig. Der Zugriff auf das SRAM ist auf die selektierte 64KByte-Page beschränkt. (Nicht im AT90S1200 implementiert.)

Datenfluß:

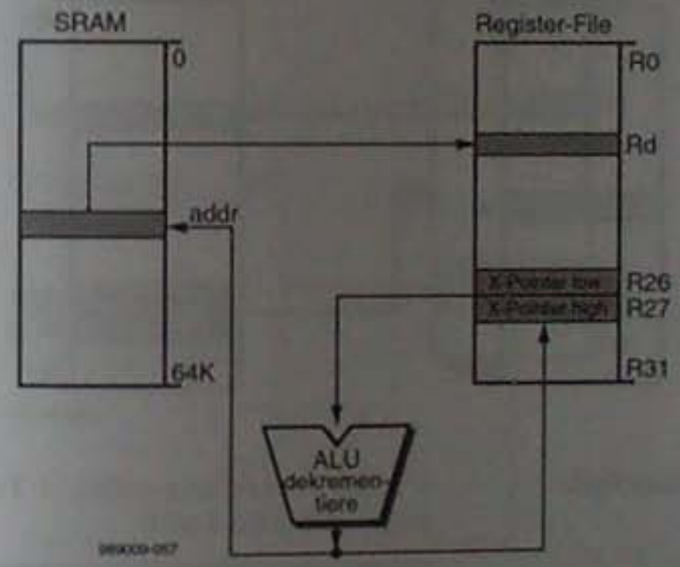


Befehlsablauf:	1 Wort, 2 Maschinenzyklen, 2 Taktimpulse: 250 ns bei 8 MHz
Flags:	I T H S V N Z C

LD Rd,–X

LD Rd,–X	Lade Register Rd indirekt über den ums eins dekrementierten X-Pointer.
Funktion:	X = X – 1; anschließend Rd ← (X)
Beschreibung:	Der Inhalt des X-Pointers (R27:R26) wird um eins dekrementiert. Anschließend wird in das Register Rd ein Byte aus dem SRAM geladen, welches durch den Wert des X-Pointer (R27:R26) adressiert wird. Der Befehl ist für alle Register R0 bis R31 im Register-File zulässig. Der Zugriff auf das SRAM ist auf die selektierte 64KByte-Page beschränkt. (Nicht im AT90S1200 implementiert.)

Datenfluß:



Befehlsablauf:	1 Wort, 2 Maschinenzyklen, 2 Taktimpulse: 250 ns bei 8 MHz
Flags:	I T H S V N Z C

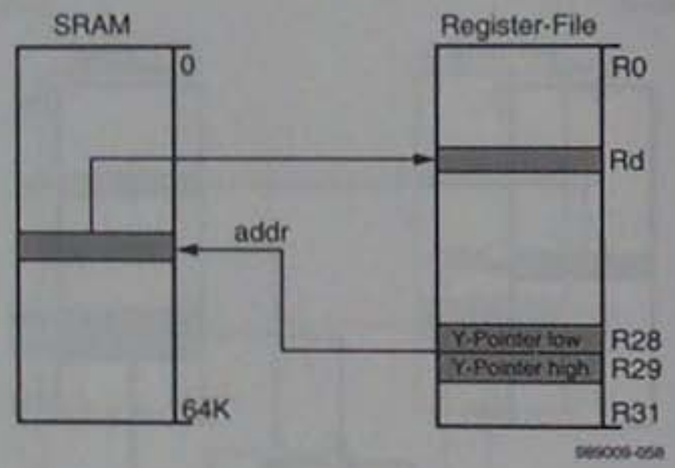
LD Rd,Y

LD Rd,Y — Lade Register Rd indirekt über Y-Pointer.

Funktion: Rd ← (Y)

Beschreibung: Lädt in das Register Rd ein Byte aus dem SRAM, welches durch den Wert des Y-Pointer (R29:R28) adressiert wird. Der Befehl ist für alle Register R0 bis R31 im Register-File zulässig. Der Zugriff auf das SRAM ist auf die selektierte 64KByte-Page beschränkt. (Nicht im AT90S1200 implementiert.)

Datenfluß:



Befehlsablauf: 1 Wort, 2 Maschinenzyklen, 2 Taktimpulse: 250 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

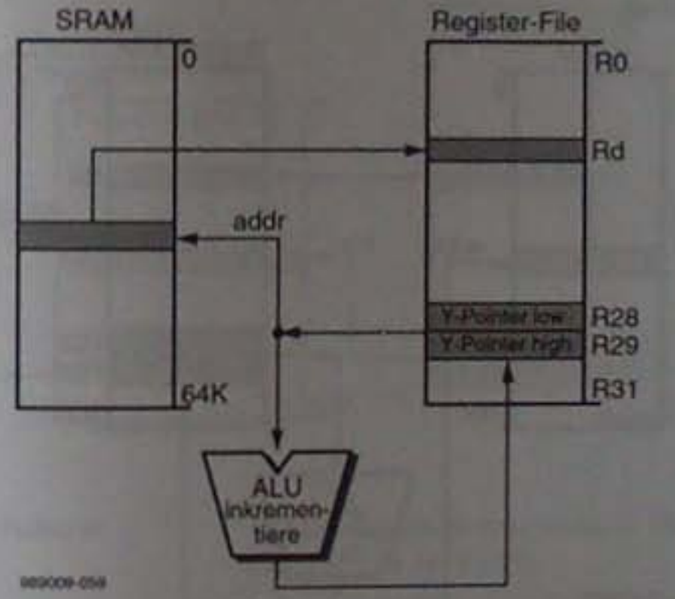
LD Rd,Y+

LD Rd,Y+ — Lade Register Rd indirekt über Y-Pointer und inkrementiere Pointer nachher.

Funktion: Rd ← (Y); anschließend Y = Y + 1

Beschreibung: Lädt in das Register Rd ein Byte aus dem SRAM, welches durch den Wert des Y-Pointer (R29:R28) adressiert wird. Anschließend wird der Y-Pointer um eins inkrementiert. Der Befehl ist für alle Register R0 bis R31 im Register-File zulässig. Der Zugriff auf das SRAM ist auf die selektierte 64KByte-Page beschränkt. (Nicht im AT90S1200 implementiert.)

Datenfluß:



Befehlsablauf: 1 Wort, 2 Maschinenzyklen, 2 Taktimpulse: 250 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

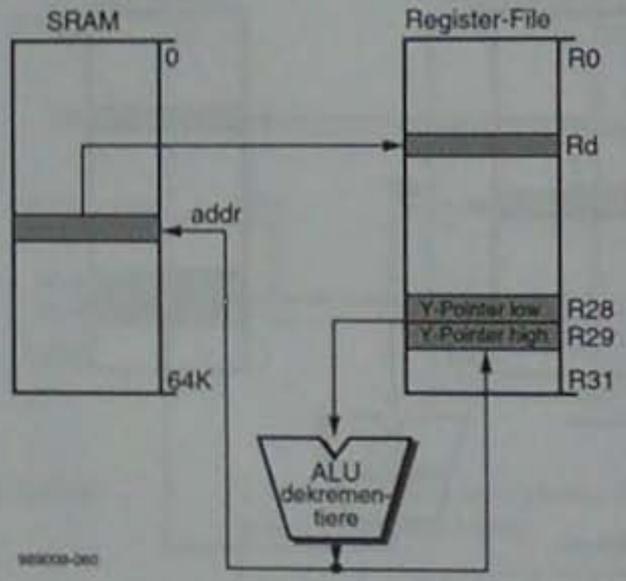
LD Rd,-Y

LD Rd,-Y — Lade Register Rd indirekt über den ums eins dekrementierten Y-Pointer.

Funktion: Y = Y - 1; anschließend Rd ← (Y)

Beschreibung: Der Inhalt des Y-Pointers (R29:R28) wird um eins dekrementiert. Anschließend wird in das Register Rd ein Byte aus dem SRAM geladen, welches durch den Wert des Y-Pointers (R29:R28) adressiert wird. Der Befehl ist für alle Register R0 bis R31 im Register-File zulässig. Der Zugriff auf das SRAM ist auf die selektierte 64KByte-Page beschränkt. (Nicht im AT90S1200 implementiert.)

Datenfluß:



Befehlsablauf: 1 Wort, 2 Maschinenzyklen, 2 Taktimpulse: 25o ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

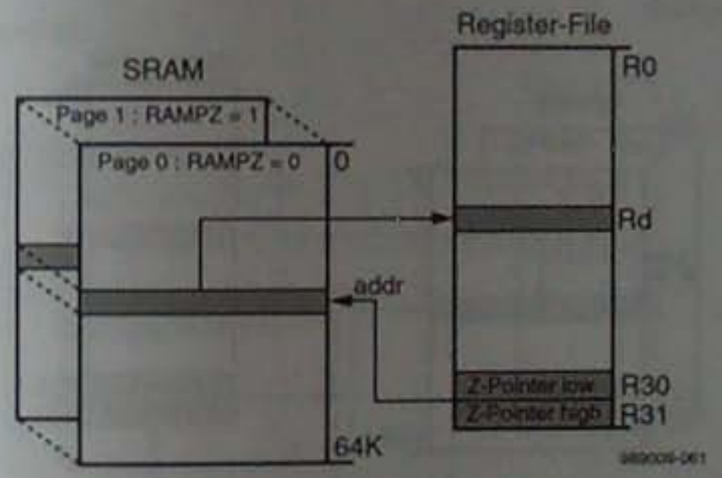
LD Rd,Z

LD Rd,Z — Lade Register Rd indirekt über Z-Pointer.

Funktion: Rd ← (Z)

Beschreibung: Lädt in das Register Rd ein Byte aus dem SRAM, welches durch den Wert des Z-Pointers (R31:R30) adressiert wird. Der Befehl ist für alle Register R0 bis R31 im Register-File zulässig. Der Zugriff auf das SRAM ist auf die selektierte 64KByte-Page beschränkt.

Datenfluß:



Befehlsablauf: 1 Wort, 2 Maschinenzyklen, 2 Taktimpulse: 250 ns bei 8 MHz

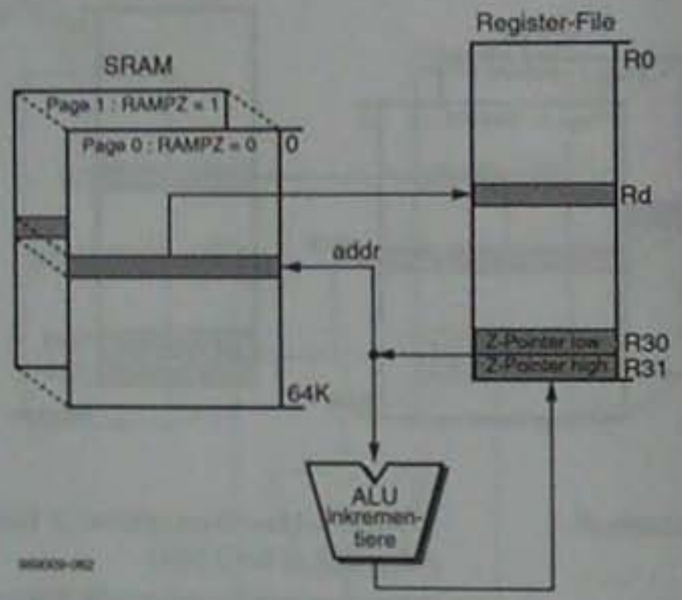
Flags:

I	T	H	S	V	N	Z	C

LD Rd,Z+

LD Rd,Z+	Lade Register Rd indirekt über Z-Pointer und inkrementiere Pointer nachher.
Funktion:	Rd ← (Z); anschließend Z = Z + 1
Beschreibung:	Lädt in das Register Rd ein Byte aus dem SRAM, welches durch den Wert des Z-Pointers (R31:R30) adressiert wird. Anschließend wird der Z-Pointer um eins inkrementiert. Der Befehl ist für alle Register R0 bis R31 im Register-File zulässig. Der Zugriff auf das SRAM ist auf die selektierte 64KByte-Page beschränkt. (Nicht im AT90S1200 implementiert.)

Datenfluß:



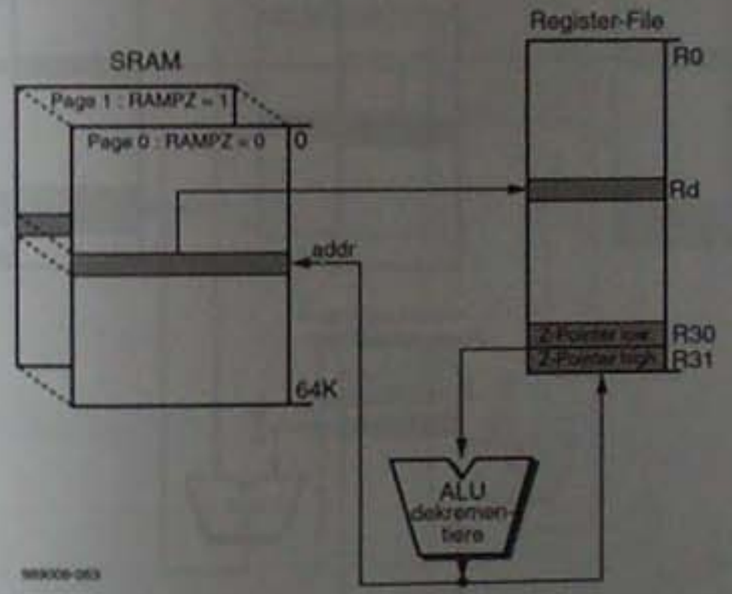
Befehlsablauf:	1 Wort, 2 Maschinenzyklen, 2 Taktimpulse: 250 ns bei 8 MHz
Flags:	

I	T	H	S	V	N	Z	C

LD Rd,-Z

LD Rd,-Z	Lade Register Rd indirekt über den ums eins dekrementierten Z-Pointer.
Funktion:	Z = Z - 1; anschließend Rd ← (Z)
Beschreibung:	Der Inhalt des Z-Pointers (R31:R30) wird um eins dekrementiert. Anschließend wird in das Register Rd ein Byte aus dem SRAM geladen, welches durch den Wert des Z-Pointers (R31:R30) adressiert wird. Der Befehl ist für alle Register R0 bis R31 im Register-File zulässig. Der Zugriff auf das SRAM ist auf die selektierte 64KByte-Page beschränkt. (Nicht im AT90S1200 implementiert.)

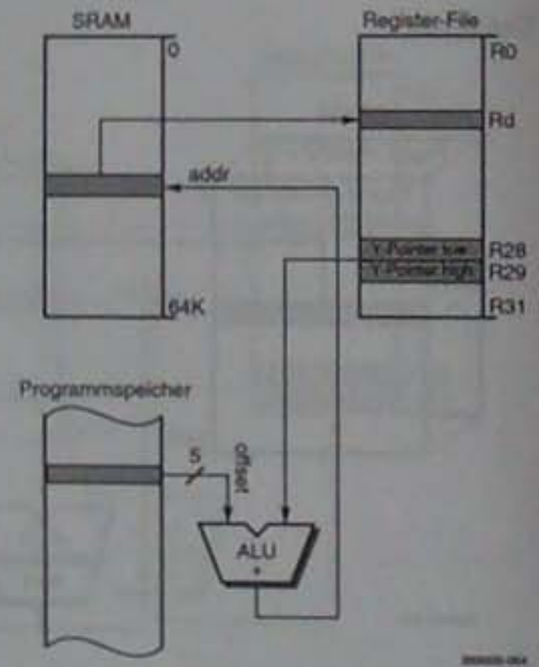
Datenfluß:



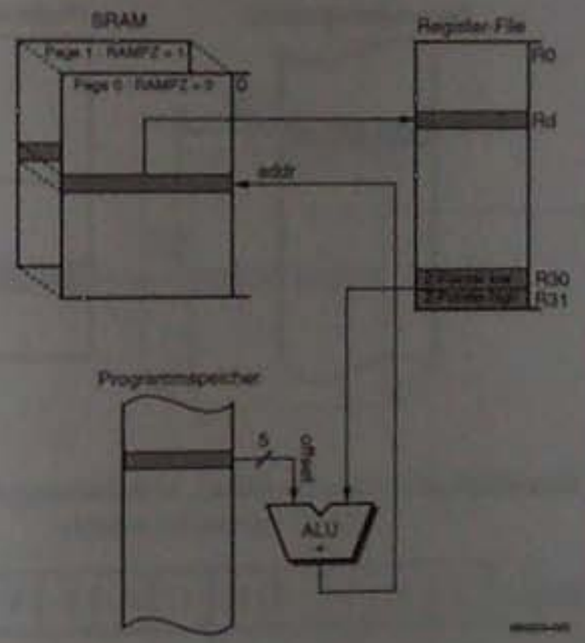
Befehlsablauf:	1 Wort, 2 Maschinenzyklen, 2 Taktimpulse: 250 ns bei 8 MHz
Flags:	

I	T	H	S	V	N	Z	C

LDD Rd, Y+ offset

LDD Rd,Y+ offset	Lade Register Rd indirekt über Y-Pointer mit Offset.
Funktion:	Rd ← (Y + offset)
Beschreibung:	Lädt in das Register Rd ein Byte aus dem SRAM, welches durch die Summe aus dem Wert des Y-Pointers (R29:R28) und dem Wert offset adressiert wird. Der Befehl ist für alle Register R0 bis R31 im Register-File zulässig. Der Wert offset darf im Bereich von 0 bis 63 (dezimal) liegen. Der Zugriff auf das SRAM ist auf die selektierte 64KByte-Page beschränkt. (Nicht im AT90S1200 implementiert.)
Datenfluß:	
Befehlsablauf:	1 Wort, 2 Maschinenzyklen, 2 Taktimpulse: 250 ns bei 8 MHz
Flags:	I T H S V N Z C

LDD Rd, Z+ offset

LDD Rd,Z+ offset	Lade Register Rd indirekt über Z-Pointer mit Offset.
Funktion:	Rd ← (Z + offset)
Beschreibung:	Lädt in das Register Rd ein Byte aus dem SRAM, welches durch die Summe aus dem Wert des Z-Pointers (R31:R30) und dem Wert offset adressiert wird. Der Befehl ist für alle Register R0 bis R31 im Register-File zulässig. Der Wert offset darf im Bereich von 0 bis 63 (dezimal) liegen. Der Zugriff auf das SRAM ist auf die selektierte 64KByte-Page beschränkt. (Nicht im AT90S1200 implementiert.)
Datenfluß:	
Befehlsablauf:	1 Wort, 2 Maschinenzyklen, 2 Taktimpulse: 250 ns bei 8 MHz
Flags:	I T H S V N Z C

LDI Rd,K

LDI Rd,K

Lade Register Rd mit dem unmittelbaren Wert K.

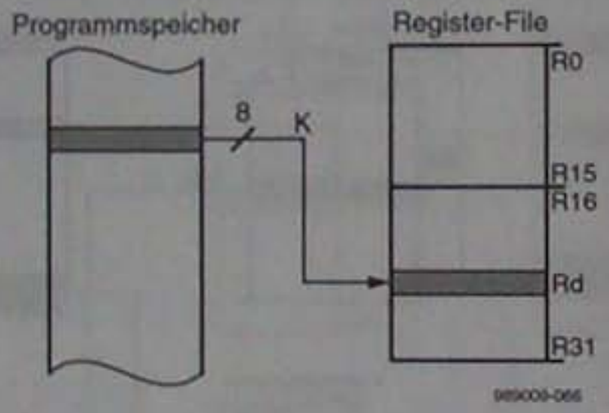
Funktion:

Rd ← K

Beschreibung:

Der unmittelbare Wert K wird in das Register Rd geladen. Der Befehl ist für alle Register R16 bis R31 in der oberen Hälfte des Register-Files zulässig. Der unmittelbare Wert K darf im Bereich von 0 bis 255 (dezimal) liegen.

Datenfluß:



Befehlsablauf:

1 Wort, 1 Maschinenzyklus, 1 Taktimpuls: 125 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

LDS Rd,addr

LDS Rd,addr

Lade Register Rd direkt mit einem Byte aus dem SRAM.

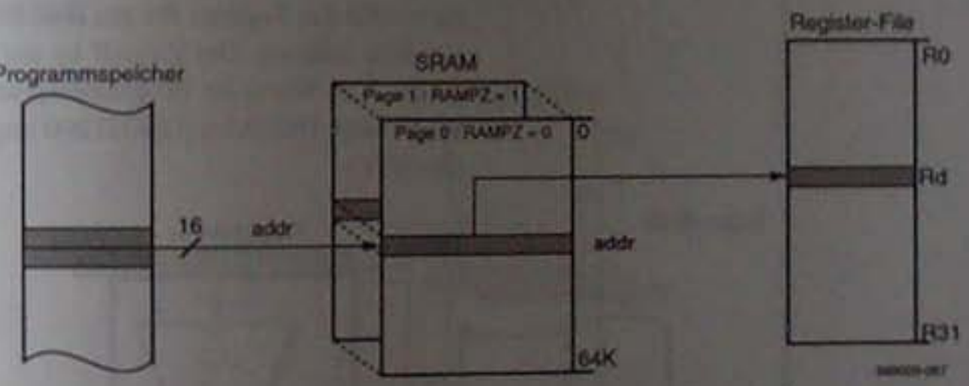
Funktion:

Rd ← SRAM(addr)

Beschreibung:

Lädt Register Rd direkt mit einem Byte aus dem SRAM, das über die Adresse addr adressiert wird. Der Befehl ist für alle Register R0 bis R31 im Register-File zulässig. Der Wert addr darf im Bereich von 0 bis 65535 (dezimal) liegen. Der Zugriff auf das SRAM ist auf die selektierte 64KByte-Page beschränkt.

Datenfluß:



Befehlsablauf:

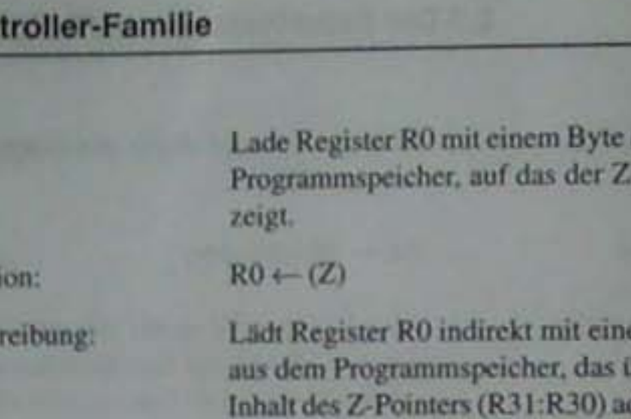
2 Worte, 3 Maschinenzyklen, 3 Taktimpulse: 375 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

LPM		
	LPM	Lade Register R0 mit einem Byte aus dem Programmspeicher, auf das der Z-Pointer zeigt.
	Funktion:	R0 ← (Z)
	Beschreibung:	Lädt Register R0 indirekt mit einem Byte aus dem Programmspeicher, das über den Inhalt des Z-Pointers (R31:R30) adressiert wird. Da der Programmspeicher 16-Bit breit ist, wird über das LSB des Z-Pointers angegeben, ob das obere oder untere Byte des Programmspeichers in das Register R0 geladen werden soll. Ist das LSB 0, so wird das untere Byte geladen. Hingegen wird bei 1 das obere Byte geladen. Der Befehl ist nur für das Register R0 aus dem Register-File zulässig. Der Zugriff ist auf die ersten 32K-Worte im Programmspeicher beschränkt. (Nicht im AT90S1200 implementiert.)

Datenfluß:



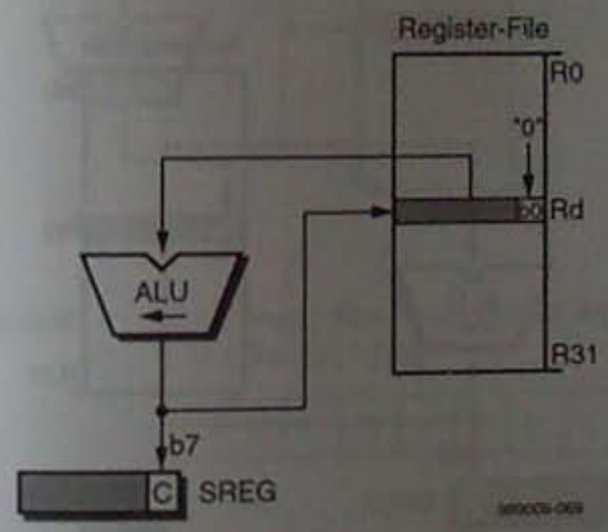
Befehlsablauf:	1 Wort, 3 Maschinenzyklen, 3 Taktimpulse: 375 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

LSL Rd		
	LSL Rd	Schiebe das Register Rd logisch nach links.
	Funktion:	C ← Rd<7>; Rd<7:1> ← Rd<6:0>; Rd<0> ← 0;
	Beschreibung:	Der Inhalt des Registers Rd wird um eine Stelle logisch nach links geschoben. Dabei wird der Inhalt des Bit 7 in das Carry-Flag geschoben. Bit 0 des Register Rd wird mit 0 überschrieben. Dieser Befehl multipliziert eine vorzeichenlose mit zwei. Der Befehl ist für alle Register R0 bis R31 im Register-File zulässig.

Datenfluß:



Befehlsablauf:	1 Wort, 1 Maschinenzyklus, 1 Taktimpuls: 125 ns bei 8 MHz

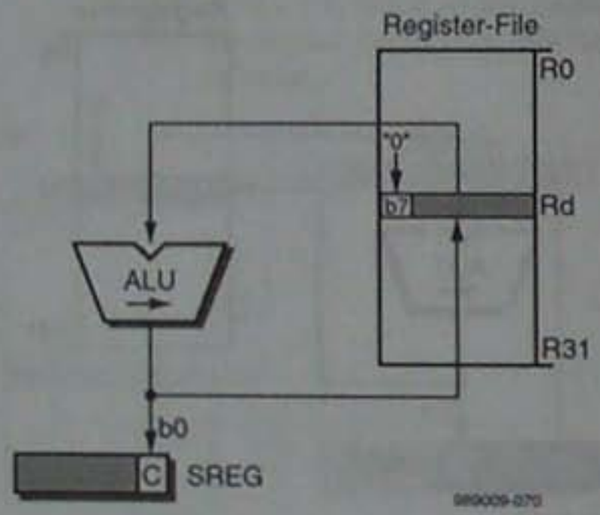
Flags:

I	T	H	S	V	N	Z	C

LSR Rd

LSR Rd	Schiebe das Register Rd logisch nach rechts.
Funktion:	C ← Rd<0>; Rd<6:0> ← Rd<7:1>; Rd<7> ← 0
Beschreibung:	Der Inhalt des Registers Rd wird um eine Stelle logisch nach rechts geschoben. Dabei wird der Inhalt des Bit 0 in das Carry-Flag geschoben. Bit 7 des Register Rd wird mit 0 überschrieben. Dieser Befehl dividiert eine vorzeichenlose durch zwei. Das Carry-Flag kann zum Runden herangezogen werden. Der Befehl ist für alle Register R0bis R31 im Register-File zulässig.

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus, 1 Taktimpuls: 125 ns bei 8 MHz

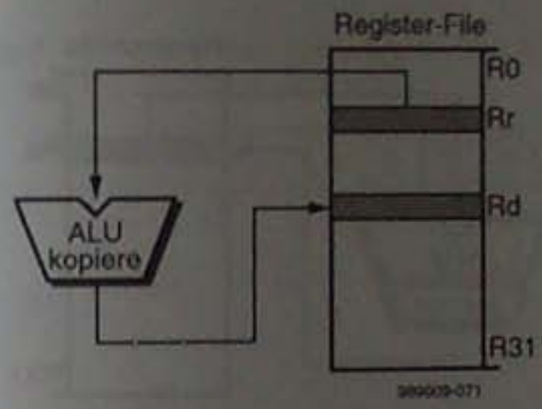
Flags:

I	T	H	S	V	N	Z	C

MOV Rd,Rr

MOV Rd,Rr	Kopiere Register Rr in Rd.
Funktion:	Rd ← Rr
Beschreibung:	Der Inhalt des Register Rr wird in das Register Rd kopiert. Der Inhalt des Registers Rr bleibt dabei unverändert. Der Befehl ist für alle Register R0 bis R31 im Register-File zulässig.

Datenfluß:



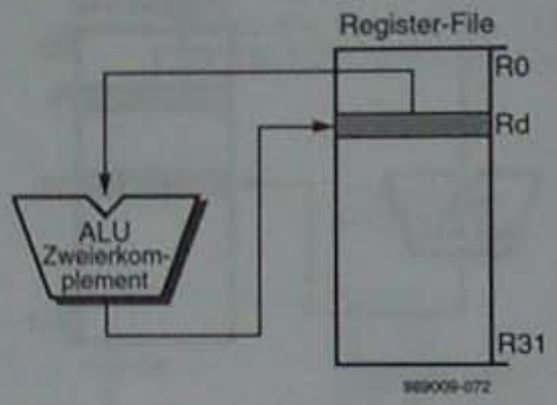
Befehlsablauf: 1 Wort, 1 Maschinenzyklus, 1 Taktimpuls: 125 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

NEG Rd

NEG Rd	Negiere Register Rd.
Funktion:	Rd ← $00 – Rd
Beschreibung:	Es wird das Zweierkomplement des Inhalts des Registers Rd gebildet. Ist der ursprüngliche Inhalt von Rd gleich $80, so wird der Inhalt nicht verändert. Der Befehl ist für alle Register R0 bis R31 im Register-File zulässig.
Datenfluß:	



Befehlsablauf:	1 Wort, 1 Maschinenzyklus, 1 Taktimpuls: 125 ns bei 8 MHz
Flags:	I T H S V N Z C

NOP

NOP	Keine Operation.
Funktion:	Verzögerung
Beschreibung:	Es wird ein funktionsloser Befehl ausgeführt, der im Programmablauf eine Verzögerung von einem Maschinenzyklus erzeugt.
Datenfluß:	
Befehlsablauf:	1 Wort, 1 Maschinenzyklus, 1 Taktimpuls: 125 ns bei 8 MHz
Flags:	I T H S V N Z C

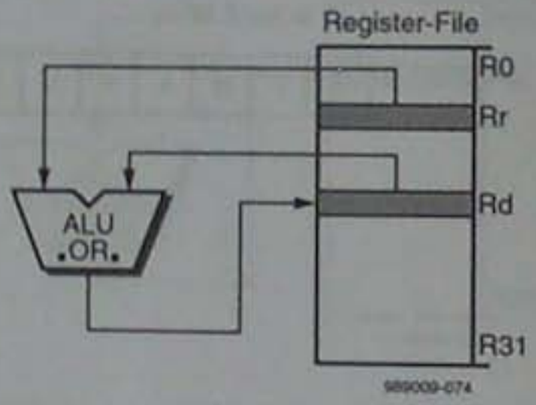
OR Rd,Rr

OR Rd,Rr — Register Rr und Rd ODER-verknüpfen.

Funktion: Rd ← Rd .OR. Rr

Beschreibung: Der Inhalt des Registers Rr wird mit dem Inhalt des Registers Rd logisch ODER-verknüpft. Das Ergebnis der Verknüpfung steht im Register Rd. Der Inhalt des Registers Rr bleibt unverändert. Der Befehl ist für alle Register R0 bis R31 im Register-File zulässig.

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus, 1 Taktimpuls: 125 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

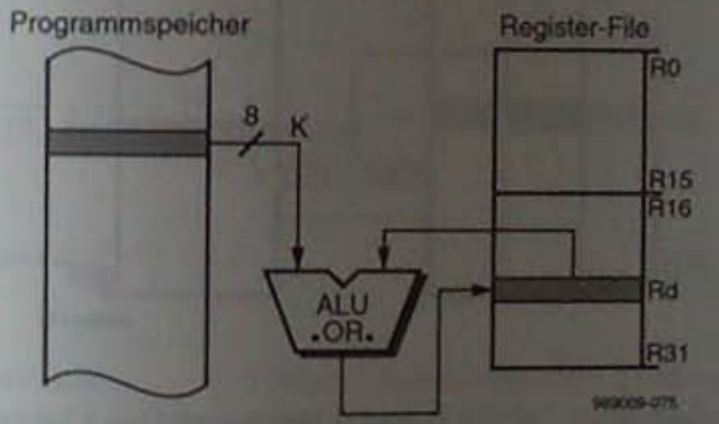
ORI Rd,K

ORI Rd,K — Unmittelbaren Wert K und Register Rd ODER-verknüpfen.

Funktion: Rd ← Rd .OR. K

Beschreibung: Der unmittelbare Wert K wird mit dem Inhalt des Registers Rd logisch ODER-verknüpft. Das Ergebnis der Verknüpfung steht im Register Rd. Der Befehl ist für alle Register R16 bis R31 in der oberen Hälfte des Register-Files zulässig. Der unmittelbare Wert K darf im Bereich von 0 bis 255 (dezimal) liegen.

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus, 1 Taktimpuls: 125 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

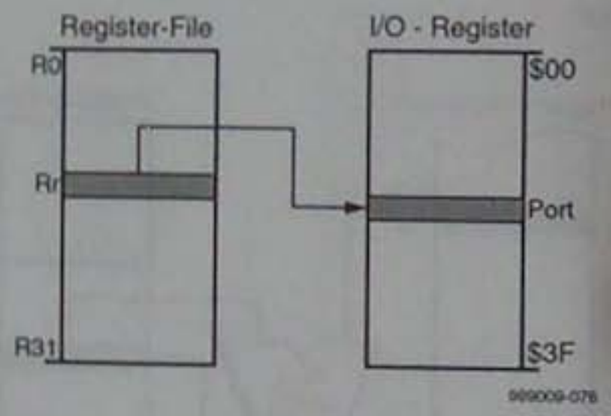
OUT Rd,Port

OUT Rd,Port — Lade Port-Register mit Rr.

Funktion: Port ← Rr

Beschreibung: Lädt in das Port-Register im I/O-Adreßraum denInhalt des Registers Rr. Der Inhalt des Registers Rr bleibt unverändert. Der Befehl ist für alle Register R0 bis R31 im Register-File zulässig. Der Wert Port kann im Bereich von 0 bis 63 (dezimal) bzw. $00 bis $3F (hex) liegen.

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus, 1 Taktimpuls: 125 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

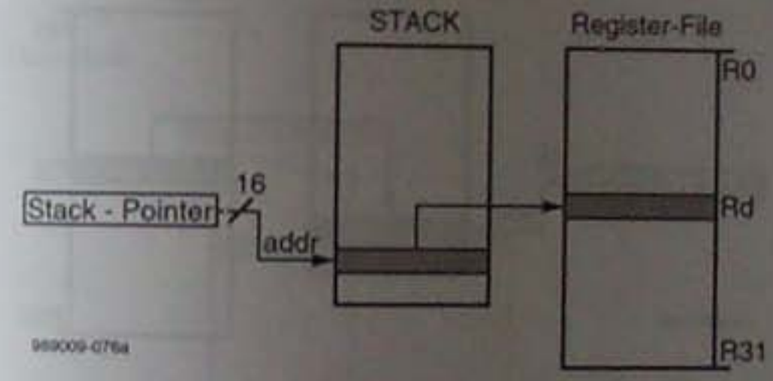
POP Rd

POP Rd — Hole Register Rd vom STACK.

Funktion: Rd ← STACK

Beschreibung: Der Inhalt der Speicherstelle, auf die der Stack-Pointer zeigt, wird in das Register Rd geladen. Anschließend wird der Stack-Pointer inkrementiert. Der Befehl ist für alle Register R0 bis R31 im Register-File zulässig. (Nicht im AT90S1200 implementiert.)

Datenfluß:



Befehlsablauf: 1 Wort, 2 Maschinenzyklen, 2 Taktimpulse; 250 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

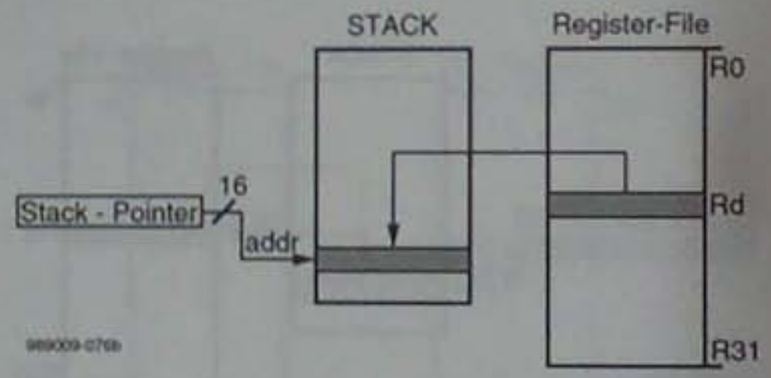
PUSH Rd

PUSH Rd — Lege Register Rd auf dem STACK ab.

Funktion: STACK ← Rd

Beschreibung: Der Stack-Pointer wird dekrementiert und der Inhalt des Registers Rd in die Speicherstelle geladen, auf die der Stack-Pointer zeigt. Der Befehl ist für alle Register R0 bis R31 im Register-File zulässig. (Nicht im AT90S1200 implementiert.)

Datenfluß:



Befehlsablauf: 1 Wort, 2 Maschinenzyklen, 2 Taktimpulse: 250 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

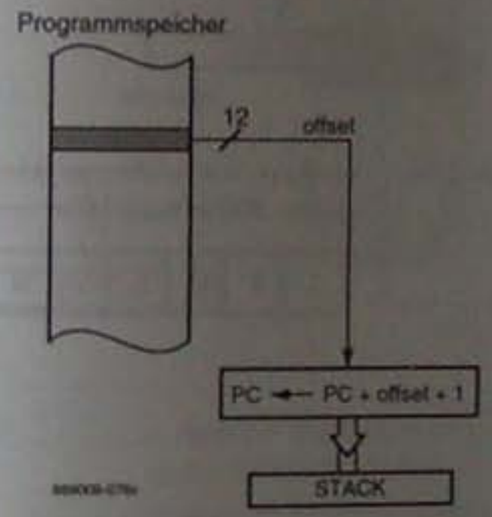
RCALL offset

RCALL offset — Aufruf eines Unterprogramms relativ zur gegenwärtigen Adresse.

Funktion: PC ← PC + 1 + offset

Beschreibung: Ein Unterprogramm mit einem Offset relativ zur gegenwärtigen Adresse wird aufgerufen. Der um eins erhöhte Programmzähler PC wird auf den Stack geschoben. Das Unterprogramm kann sich von der gegenwärtigen Adresse im Programmzähler um den Wert offset (12 Bit) von –2K-Worte bis +2K-Worte entfernt im Programmspeicher befinden.

Datenfluß:



Befehlsablauf: 1 Wort, 3 Maschinenzyklen, 3 Taktimpulse: 375 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

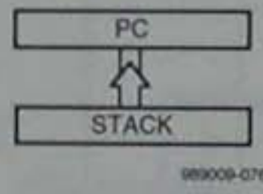
RET

RET — Rücksprung vom Unterprogramm.

Funktion: PC ← STACK

Beschreibung: Der Inhalt der Speicherstelle, auf die der Stack-Pointer zeigt, wird in den Programmzähler PC geladen. Anschließend wird der Stack-Pointer um zwei inkrementiert. Die Befehlsausführung wird an der Adresse fortgeführt, auf die der Programmzähler zeigt.

Datenfluß:



Befehlsablauf: 1 Wort, 4 Maschinenzyklen, 4 Taktimpulse: 500 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

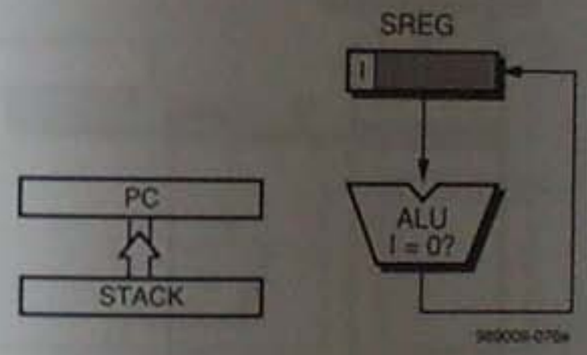
RETI

RETI — Rücksprung vom Interrupt.

Funktion: PC ← STACK

Beschreibung: Der Inhalt der Speicherstelle, auf die der Stack-Pointer zeigt, wird in den Programmzähler PC geladen. Anschließend wird der Stack-Pointer um zwei inkrementiert. Die Befehlsausführung wird an der Adresse fortgeführt, auf die der Programmzähler zeigt. Ferner wird das Global-Interrupt-Flag im STATUS-Register gelöscht.

Datenfluß:



Befehlsablauf: 1 Wort, 4 Maschinenzyklen, 4 Taktimpulse: 500 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

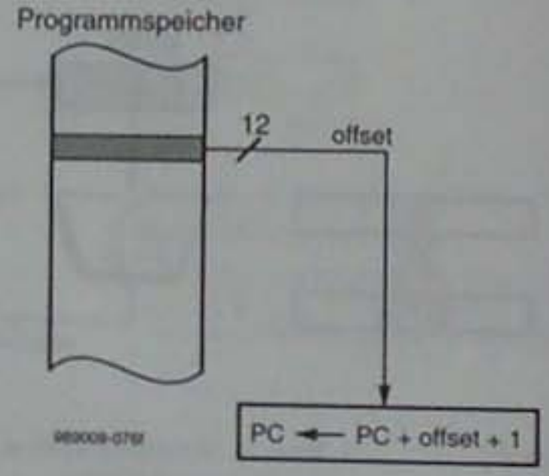
RJMP offset

RJMP offset — Sprung relativ zur gegenwärtigen Adresse.

Funktion: PC ← PC + 1 + offset

Beschreibung: Ein Sprung mit einem Offset relativ zur gegenwärtigen Adresse wird ausgeführt. Zum Inhalt des Programmzählers wird der Wert offset +1 addiert. Von der gegenwärtigen Adresse im Programmzähler kann um einen Offset von −2K-Worte bis +2K-Worte entfernt im Programmspeicher gesprungen werden.

Datenfluß:



Befehlsablauf: 1 Wort, 3 Maschinenzyklen, 3 Taktimpulse: 375 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

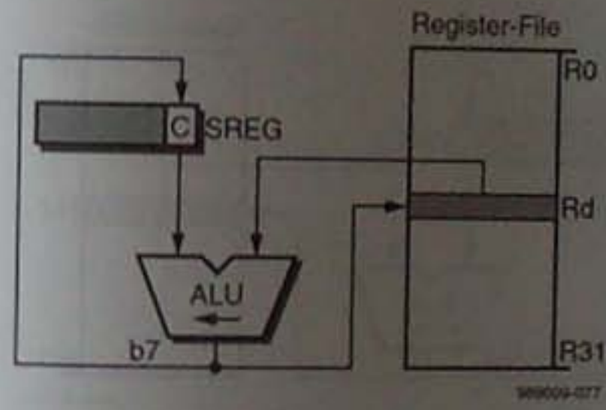
ROL Rd

ROL Rd — Rotiere das Register Rd nach links.

Funktion: C ← Rd<7>; Rd<7:1> ← Rd<6:0>; Rd<0> ← C;

Beschreibung: Der Inhalt des Registers Rd wird um eine Stelle nach links rotiert. Die Rotation des Registers Rd erfolgt über das Carry-Flag. Dabei wird Bit 0 des Registers Rd mit dem Carry-Flag geladen und der Inhalt des Bit 7 in das Carry-Flag geschoben. Der Befehl ist für alle Register R0 bis R31 im Register-File zulässig.

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus, 1 Taktimpuls: 125 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

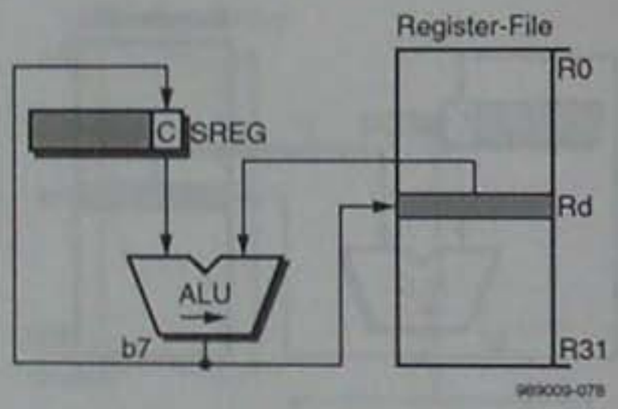
ROR Rd

ROR Rd — Rotiere das Register Rd nach links.

Funktion: C ← Rd<0>; Rd<6:0> ← Rd<7:1>; Rd<7> ← C

Beschreibung: Der Inhalt des Registers Rd wird um eine Stelle nach rechts rotiert. Die Rotation des Registers Rd erfolgt über das Carry-Flag. Dabei wird Bit 7 des Registers Rd mit dem Carry-Flag geladen und der Inhalt des Bit 0 in das Carry-Flag geschoben. Der Befehl ist für alle Register R0 bis R31 im Register-File zulässig.

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus, 1 Taktimpuls: 125 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

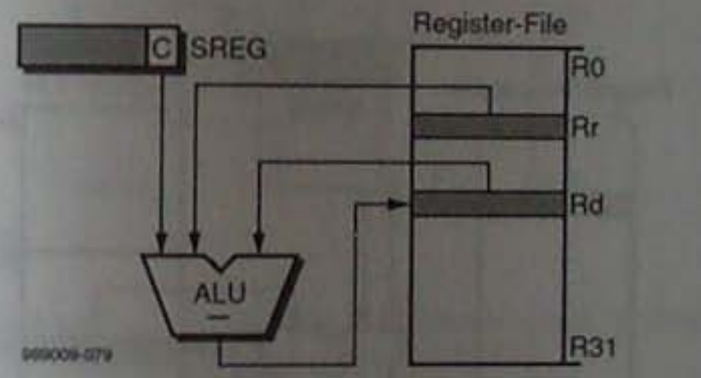
SBC Rd,Rr

SBC Rd,Rr — Subtrahiere Register Rr und Carry-Flag vom Register Rd

Funktion: Rd ← Rd – Rr – C

Beschreibung: Der Inhalt des Registers Rr und das Carry-Flag werden vom Inhalt des Registers Rd subtrahiert. Das Ergebnis der Subtraktion steht im Register Rd. Der Inhalt des Registers Rr bleibt unverändert. Der Befehl ist für alle Register R0 bis R31 im Register-File zulässig.

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus, 1 Taktimpuls: 125 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

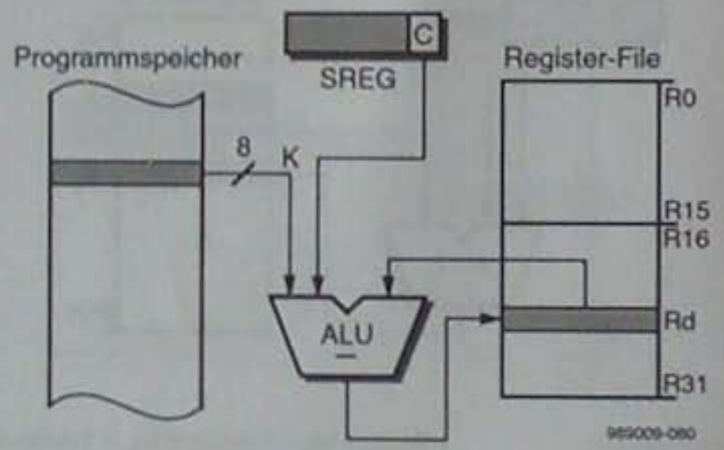
SBCI Rd,K

SBCI Rd,K — Subtrahiere unmittelbaren Wert K und Carry-Flag vom Register Rd

Funktion: Rd ← Rd – K – C

Beschreibung: Der unmittelbare Wert K und das Carry-Flag werden vom Inhalt des Registers Rd subtrahiert. Das Ergebnis der Subtraktion steht im Register Rd. Der Befehl ist für alle Register R16 bis R31 in der oberen Hälfte des Register-Files zulässig. Der unmittelbare Wert K kann im Bereich von 0 bis 255 (dezimal) liegen.

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus, 1 Taktimpuls: 125 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

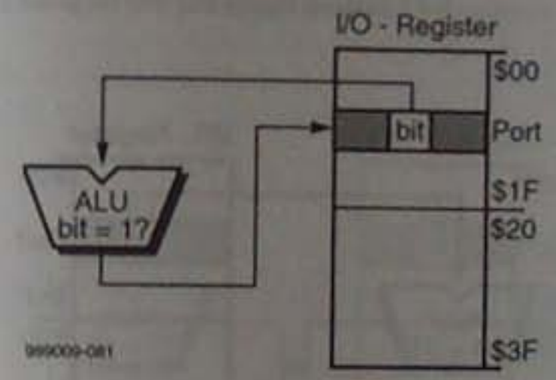
SBI Port,bit

SBI Port,bit — Setze ein Bit im Port-Register

Funktion: Port<bit> ← 1

Beschreibung: Setzt das angegebene Bit in einem Port-Register. Der Wert bit kann im Bereich von 0 bis 7 liegen. Der Wert Port kann im Bereich von 0 bis 31 ($00 bis $1F) liegen.

Datenfluß:



Befehlsablauf: 1 Wort, 2 Maschinenzyklen, 2 Taktimpulse: 250 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

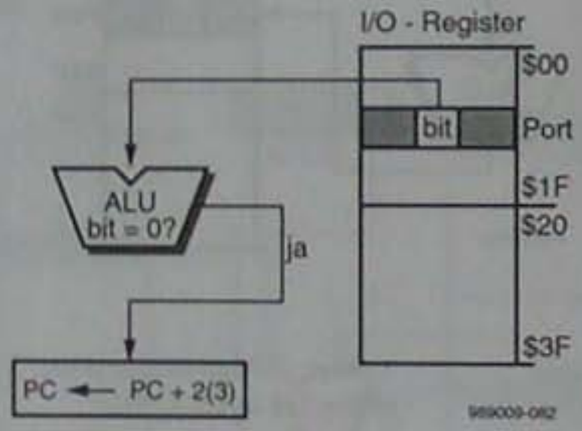
SBIC Port,bit

SBIC Port,bit — Springe falls Bit im Port-Register gelöscht.

Funktion: PC ← PC + 2, falls Port<bit> = 0

Beschreibung: Der nächste Befehl im Programmspeicher wird übersprungen, falls das Bit „bit" im Port-Register gelöscht ist. Andernfalls wird die Programmausführung normal fortgesetzt. Der Wert bit kann im Bereich von 0 bis 7 liegen. Der Wert Port kann im Bereich von 0 bis 31 ($00 bis $1F) liegen (untere Hälfte der I/O-Register).

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus (2 bei Sprung), 1 Taktimpuls (2 bei Sprung): 125 ns (250 ns bei Sprung) bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

Anmerkung: Der nachfolgende Befehl, der übersprungen werden soll, muß ein Ein-Wort-Befehl sein, da sonst der Sprung in den Befehl selber stattfindet und somit ein unvorhersehbarer Fehler auftritt.

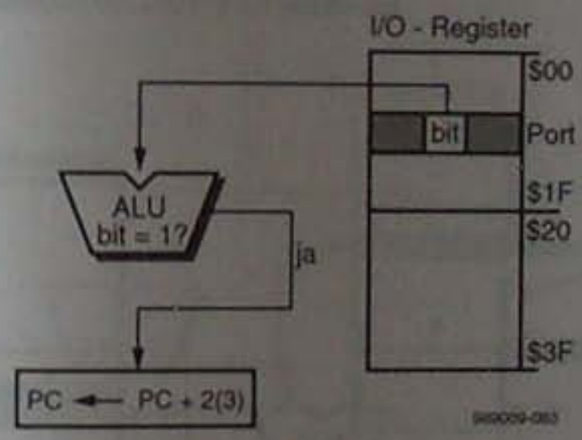
SBIS Port,bit

SBIS Port,bit — Springe falls Bit im Port-Register gesetzt.

Funktion: PC ← PC + 2, falls Port<bit> = 1

Beschreibung: Der nächste Befehl im Programmspeicher wird übersprungen, falls das Bit „bit" im Port-Register gesetzt ist. Andernfalls wird die Programmausführung normal fortgesetzt. Der Wert bit kann im Bereich von 0 bis 7 liegen. Der Wert Port kann im Bereich von 0 bis 31 ($00 bis $1F) liegen (untere Hälfte der I/O-Register).

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus (2 bei Sprung), 1 Taktimpuls (2 bei Sprung): 125 ns (250 ns bei Sprung) bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

Anmerkung: Der nachfolgende Befehl, der übersprungen werden soll, muß ein Ein-Wort-Befehl sein, da sonst der Sprung in den Befehl selber stattfindet und somit ein unvorhersehbarer Fehler auftritt.

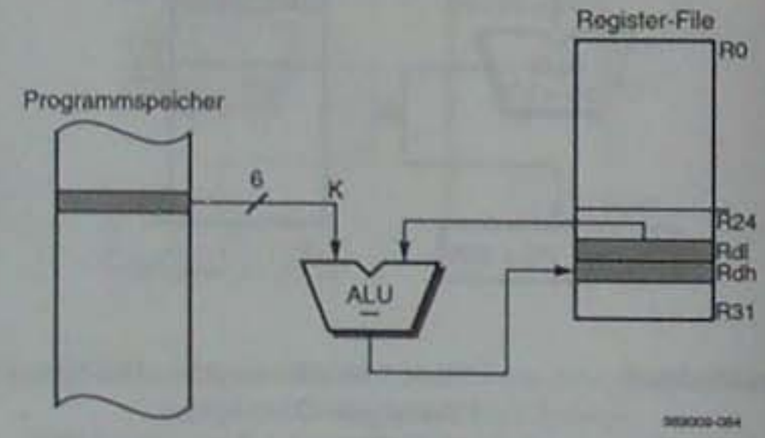
SBIW Rdl,K

SBIW Rdl,K — Subtrahiere unmittelbaren Wert K vom Registerpaar Rdh:Rdl.

Funktion: Rdh:Rdl ← Rdh:Rdl – K

Beschreibung: Der unmittelbare Wert K (K im Bereich von 0 bis 63) wird vom Inhalt des Registerpaares Rdh:Rdl subtrahiert. Dabei gibt Rdl das untere der beiden Register an. Gültige Werte für Rdl sind 24, 26, 28 und 30. Der Wert für Rdh ist dann automatisch 25, 27, 29 oder 31. Dieser Befehl bezieht sich also nur auf die oberen vier Registerpaare und ist somit für Zeigerbefehle bestens geeignet. (Nicht im AT90S1200 implementiert.)

Datenfluß:



Befehlsablauf: 1 Wort, 2 Maschinenzyklen, 2 Taktimpulse: 250 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

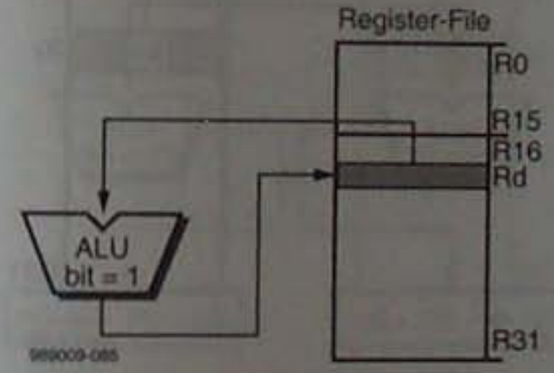
SBR Rd,mask

SBR Rd,mask — Setze Bit im Register Rd.

Funktion: Rd ← Rd .OR. – mask

Beschreibung: Setzt ein Bit im Register Rd. Das zu setzende Bit wird mit dem Wert mask maskiert. Soll z. B. das Bit 3 im gesetzt werden, so muß mask den Wert 00001000b (binär) bzw. $08 (hex) annehmen. Der Befehl ist für die Register R16 bis R31 in der oberen Hälfte des Register-Files zulässig. Der Wert mask kann im Bereich von 0 bis 255 (dezimal) liegen.

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus, 1 Taktimpuls: 125 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

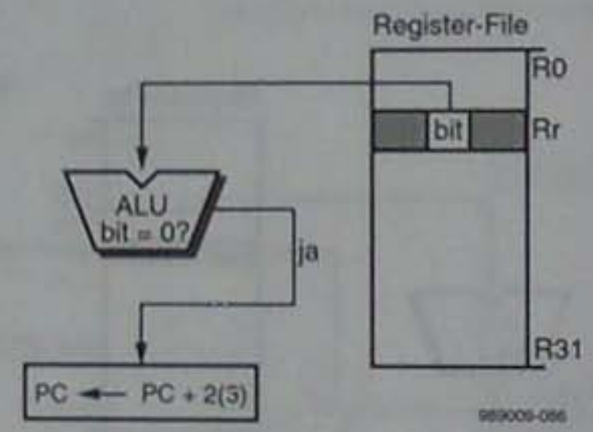
SBRC Rr,bit

SBRC Rr,bit — Springe falls Bit im Register Rr gelöscht.

Funktion: PC ← PC + 2, falls Rr<bit> = 0

Beschreibung: Der nächste Befehl im Programmspeicher wird übersprungen, falls das Bit „bit" im Register Rr gelöscht ist. Andernfalls wird die Programmausführung normal fortgesetzt. Der Wert bit kann im Bereich von 0 bis 7 liegen. Der Befehl ist für alle Register R0 bis R31 im Register-File zulässig.

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus (2 bei Sprung), 1 Taktimpuls (2 bei Sprung): 125 ns (250 ns bei Sprung) bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

Anmerkung: Der nachfolgende Befehl, der übersprungen werden soll, muß ein Ein-Wort-Befehl sein, da sonst der Sprung in den Befehl selber stattfindet und somit ein unvorhersehbarer Fehler auftritt.

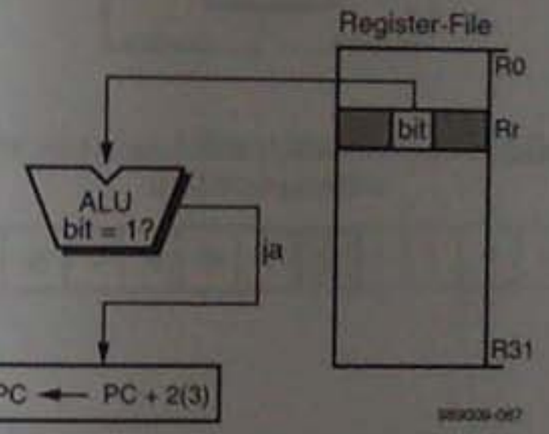
SBRS Rr,bit

SBRS Rr,bit — Springe falls Bit im Register Rr gesetzt.

Funktion: PC ← PC + 2, falls Rr<bit> = 1

Beschreibung: Der nächste Befehl im Programmspeicher wird übersprungen, falls das Bit „bit" im Register Rr gesetzt ist. Andernfalls wird die Programmausführung normal fortgesetzt. Der Wert bit kann im Bereich von 0 bis 7 liegen. Der Befehl ist für alle Register R0 bis R31 im Register-File zulässig.

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus (2 bei Sprung), 1 Taktimpuls (2 bei Sprung): 125 ns (250 ns bei Sprung) bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

Anmerkung: Der nachfolgende Befehl, der übersprungen werden soll, muß ein Ein-Wort-Befehl sein, da sonst der Sprung in den Befehl selber stattfindet und somit ein unvorhersehbarer Fehler auftritt.

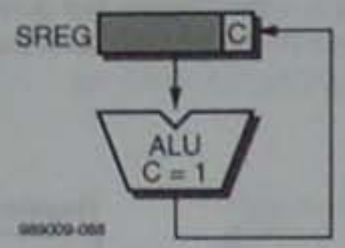
SEC

SEC — Setze Carry.

Funktion: C ← 1

Beschreibung: Das Carry-Flag im STATUS-Register SREG wird gesetzt.

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus, 1 Taktimpuls: 125 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C
							C

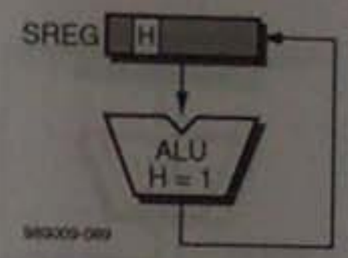
SEH

SEH — Setze Half-Carry-Flag.

Funktion: H ← 1

Beschreibung: Das Half-Carry-Flag im STATUS-Register SREG wird gesetzt.

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus, 1 Taktimpuls: 125 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C
		H					

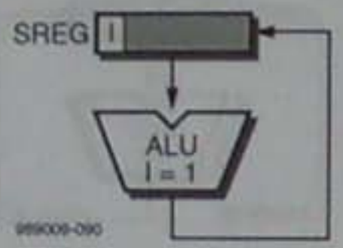
SEI

SEI — Setze Interrupt-Flag.

Funktion: I ← 1

Beschreibung: Das Global-Intterupt-Flag im STATUS-Register SREG wird gesetzt.

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus, 1 Taktimpuls: 125 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

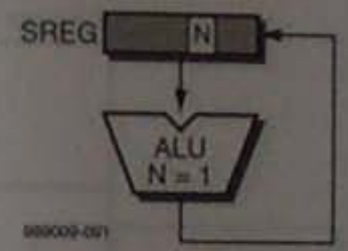
SEN

SEN — Setze Negative-Flag.

Funktion: N ← 1

Beschreibung: Das Negative-Flag im STATUS-Register SREG wird gesetzt.

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus, 1 Taktimpuls: 125 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

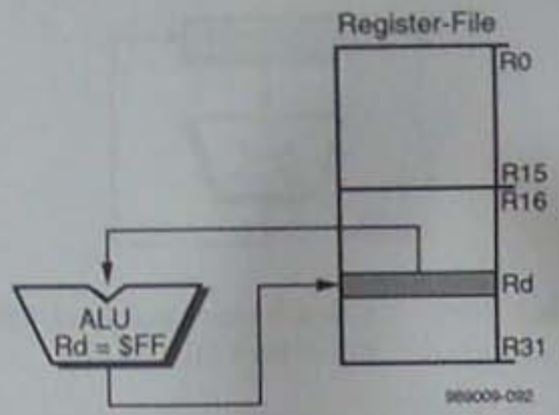
SER Rd

SER Rd — Setze alle Bits im Register Rd.

Funktion: Rd ← $FF

Beschreibung: Setze alle Bits des Register Rd. Der Befehl ist für die Register R16 bis R31 in der oberen Hälfte des Register-Files zulässig.

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus, 1 Taktimpuls: 125 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

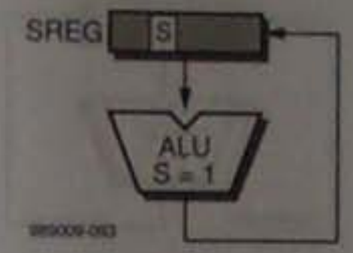
SES

SES — Setze Signed-Flag.

Funktion: S ← 1

Beschreibung: Das Signed-Flag im STATUS-Register SREG wirdgesetzt.

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus, 1 Taktimpuls: 125 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

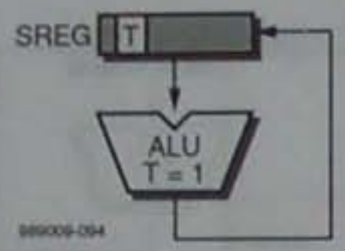
SET

SET — Setze Transfer-Bit.

Funktion: T ← 1

Beschreibung: Das Transfer-Bit im STATUS-Register SREG wird gesetzt.

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus, 1 Taktimpuls: 125 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

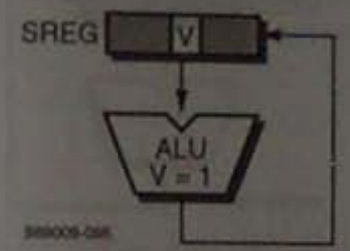
SEV

SEV — Setze V-Flag.

Funktion: V ← 1

Beschreibung: Der Zweierkomplement-Überlauf-Indikator im STATUS-Register SREG wird gesetzt.

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus, 1 Taktimpuls: 125 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

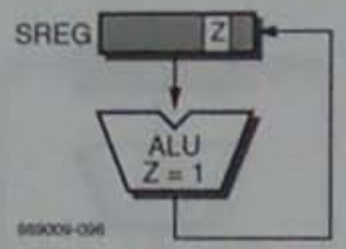
SEZ

SEZ — Setze Zero-Flag.

Funktion: Z ← 1

Beschreibung: Das Zero-Flag im STATUS-Register SREG wird gesetzt.

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus, 1 Taktimpuls: 125 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

SLEEP

SLEEP — Setze Controller in SLEEP-Modus.

Funktion:

Beschreibung: Setzt den AVR-Controller in den SLEEP-Modus, der im MCU-Control-Register MCUCR definiert ist. Wenn ein Interrupt den Controller aus dem SLEEP-Modus holt, wird zunächst der dem SLEEP-Befehl folgende Befehl ausgeführt, bevor der Interrupt-Handler gestartet wird.

Datenfluß:

Befehlsablauf: 1 Wort, 1 Maschinenzyklus, 1 Taktimpuls: 125 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

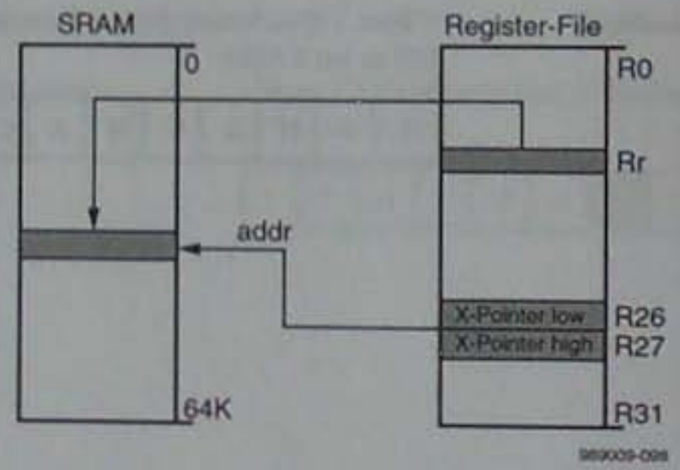
ST X,Rr

ST X,Rr — Speichere Register Rr indirekt über X-Pointer.

Funktion: (X) ← Rr

Beschreibung: Speichert den Inhalt des Registers Rr in ein Byte des SRAM, welches durch den Wert des X-Pointers (R27:R26) adressiert wird. Der Befehl ist für alle Register R0 bis R31 im Register Files zulässig. Der Zugriff auf das SRAM ist auf die selektierte 64KByte-Page beschränkt. (Nicht im AT90S1200 implementiert.)

Datenfluß:



Befehlsablauf: 1 Wort, 2 Maschinenzyklen, 2 Taktimpulse: 250 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

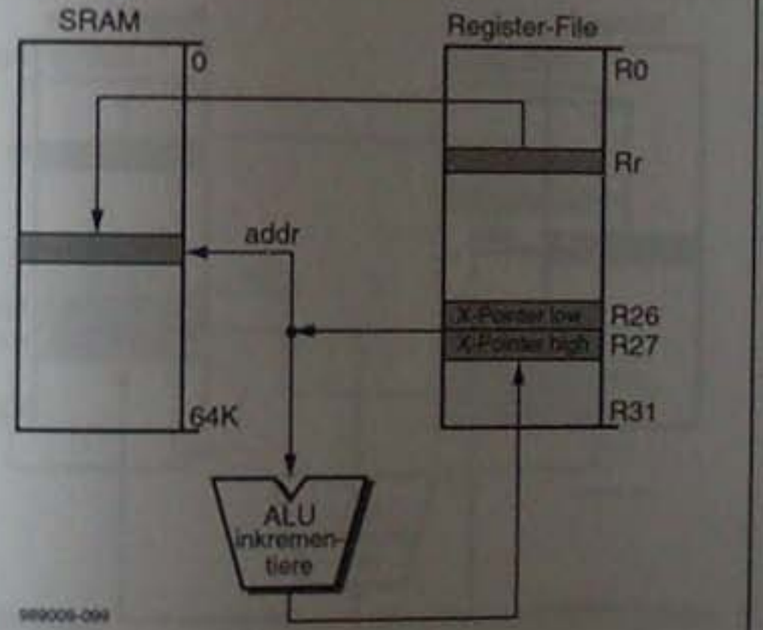
ST X+,Rr

ST X+,Rr — Speichere Register Rr indirekt über X-Pointer und inkrementiere Pointer nachher.

Funktion: (X) ← Rr; anschließend X = X + 1

Beschreibung: Speichert den Inhalt des Registers Rr in ein Byte des SRAM, welches durch den Wert des X-Pointers (R27:R26) adressiert wird. Anschließend wird der X-Pointer inkrementiert. Der Befehl ist füralle Register R0 bis R31 im Register-File zulässig. Der Zugriff auf das SRAM ist auf die selektierte 64KByte-Page beschränkt. (Nicht im AT90S1200 implementiert.)

Datenfluß:



Befehlsablauf: 1 Wort, 2 Maschinenzyklen, 2 Taktimpulse: 250 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

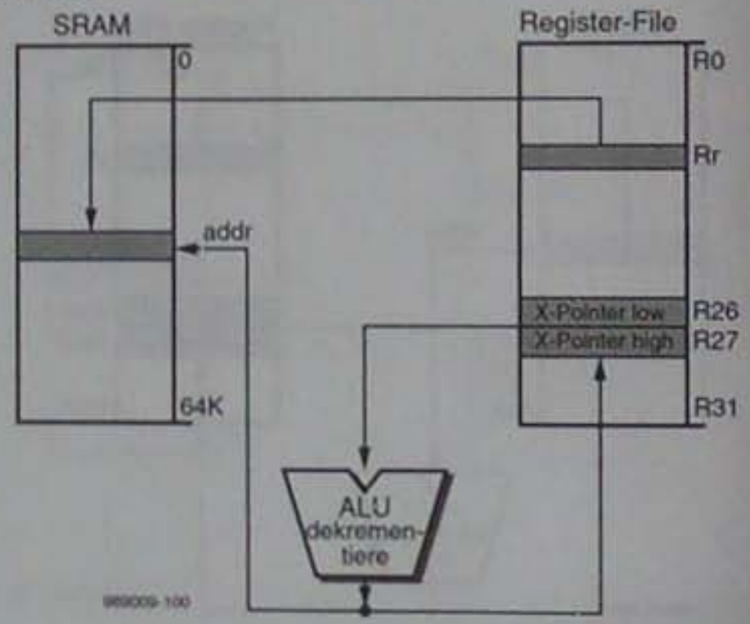
ST –X,Rr

ST –X,Rr — Speichere Register Rr indirekt über den um eins dekrementierten X-Pointer.

Funktion: X = X – 1; anschließend (X) ← Rr

Beschreibung: Speichert den Inhalt des Registers Rr in ein Byte des SRAM, welches durch den Wert des X-Pointers (R27:R26) adressiert wird. Anschließend wird der X-Pointer inkrementiert. Der Befehl ist für alle Register R0 bis R31 im Register-File zulässig. Der Zugriff auf das SRAM ist auf die selektierte 64KByte-Page beschränkt. (Nicht im AT90S1200 implementiert.)

Datenfluß:



Befehlsablauf: 1 Wort, 2 Maschinenzyklen, 2 Taktimpulse: 250 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

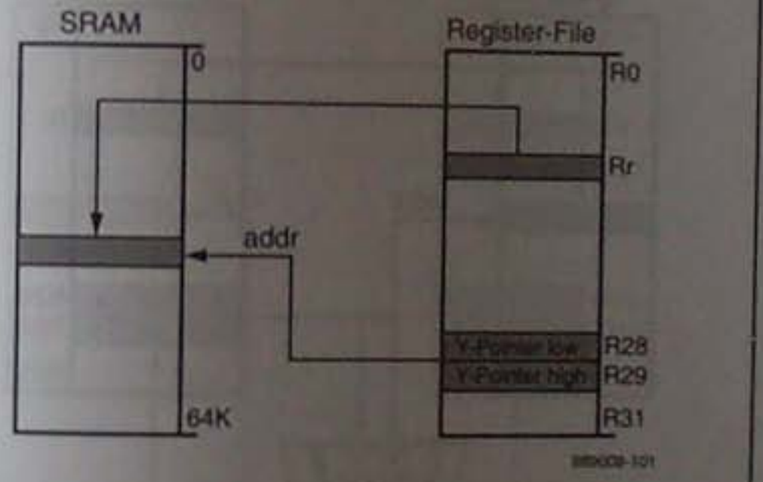
ST Y,Rr

ST Y,Rr — Speichere Register Rr indirekt über Y-Pointer.

Funktion: (Y) ← Rr

Beschreibung: Speichert den Inhalt des Registers Rr in ein Byte des SRAM, welches durch den Wert des Y-Pointers (R29:R28) adressiert wird. Der Befehl ist für alle Register R0 bis R31 im RegisterFiles zulässig. Der Zugriff auf das SRAM ist auf die selektierte 64KByte-Page beschränkt. (Nicht im AT90S1200 implementiert.)

Datenfluß:



Befehlsablauf: 1 Wort, 2 Maschinenzyklen, 2 Taktimpulse: 250 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

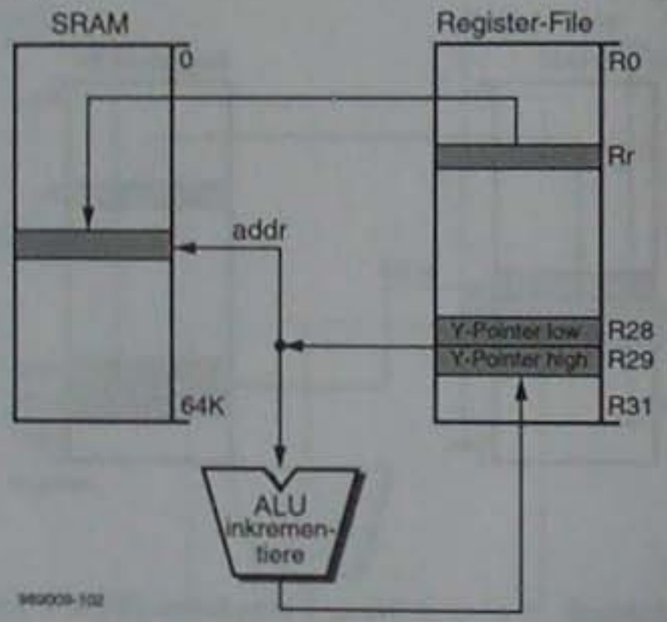
ST Y+,Rr

ST Y+,Rr — Speichere Register Rr indirekt über Y-Pointer und inkrementiere Pointer nachher.

Funktion: (Y) ← Rr; anschließend Y = Y + 1

Beschreibung: Speichert den Inhalt des Registers Rr in ein Byte des SRAM, welches durch den Wert des Y-Pointers (R29:R28) adressiert wird. Anschließend wird der Y-Pointer inkrementiert. Der Befehl ist für alle Register R0 bis R31 im Register-File zulässig. Der Zugriff auf das SRAM ist auf die selektierte 64KByte-Page beschränkt. (Nicht im AT90S1200 implementiert.)

Datenfluß:



Befehlsablauf: 1 Wort, 2 Maschinenzyklen, 2 Taktimpulse: 250 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

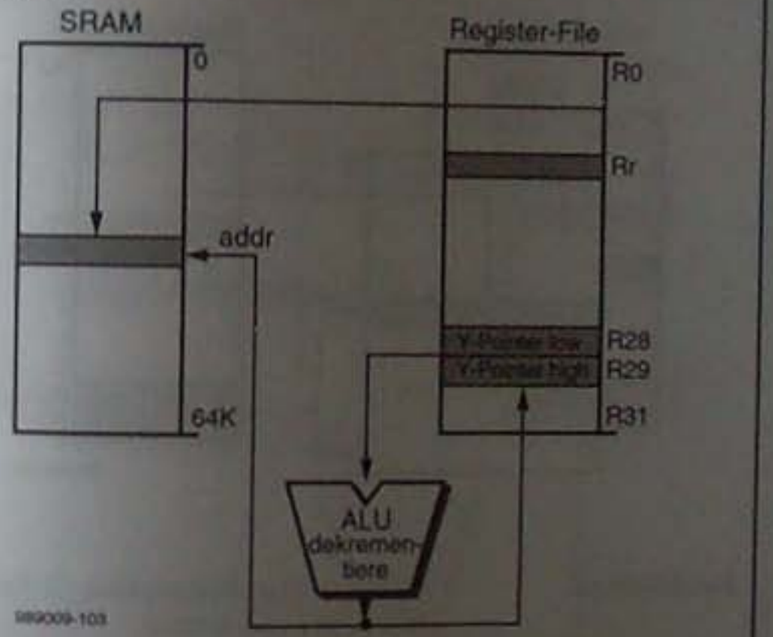
ST –Y,Rr

ST –Y,Rr — Speichere Register Rr indirekt über den um eins dekrementierten Y-Pointer.

Funktion: Y = Y – 1; anschließend (Y) ← Rr

Beschreibung: Speichert den Inhalt des Registers Rr in ein Byte des SRAM, welches durch den Wert des Y-Pointers (R29:R28) adressiert wird. Anschließend wird der Y-Pointer inkrementiert. Der Befehl ist für alle Register R0 bis R31 im Register-File zulässig. Der Zugriff auf das SRAM ist auf die selektierte 64KByte-Page beschränkt. (Nicht im AT90S1200 implementiert.)

Datenfluß:



Befehlsablauf: 1 Wort, 2 Maschinenzyklen, 2 Taktimpulse: 250 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

ST Z,Rr

ST Z,Rr

Speichere Register Rr indirekt über Z-Pointer.

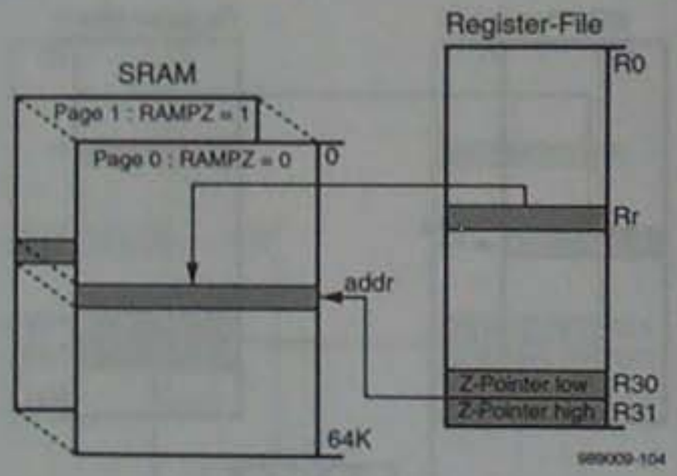
Funktion:

(Z) ← Rr

Beschreibung:

Speichert den Inhalt des Registers Rr in ein Byte des SRAM, welches durch den Wert des Z-Pointers (R31:R30) adressiert wird. Der Befehl ist für alle Register R0 bis R31 im RegisterFiles zulässig. Der Zugriff auf das SRAM ist auf die selektierte 64KByte-Page beschränkt.

Datenfluß:



Befehlsablauf:

1 Wort, 2 Maschinenzyklen, 2 Taktimpulse: 250 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

ST Z+,Rr

ST Z+,Rr

Speichere Register Rr indirekt über Z-Pointer und inkrementiere Pointer nachher.

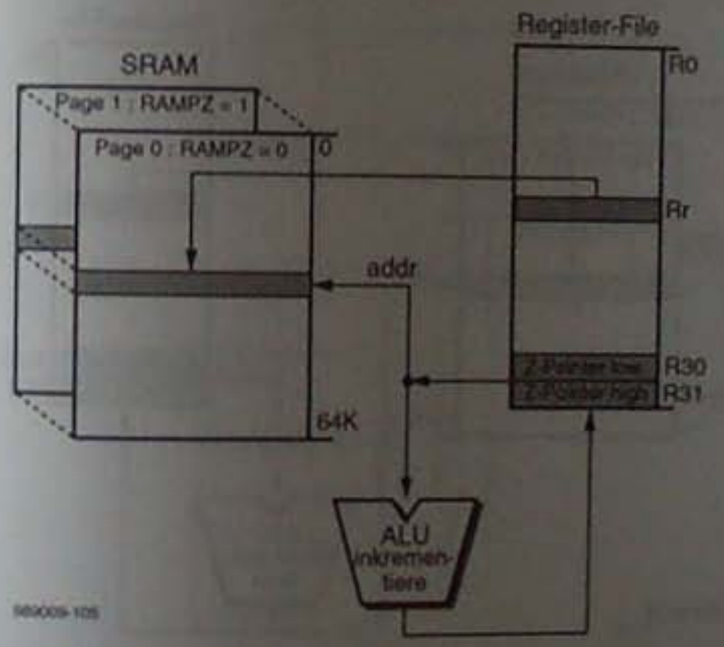
Funktion:

(Z) ← Rr; anschließend Z = Z + 1

Beschreibung:

Speichert den Inhalt des Registers Rr in ein Byte des SRAM, welches durch den Wert des Z-Pointers (R31:R30) adressiert wird. Anschließend wird der Z-Pointer inkrementiert. Der Befehl ist für alle Register R0 bis R31 im Register-File zulässig. Der Zugriff auf das SRAM ist auf die selektierte 64KByte-Page beschränkt. (Nicht im AT90S1200 implementiert.)

Datenfluß:



Befehlsablauf:

1 Wort, 2 Maschinenzyklen, 2 Taktimpulse: 250 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

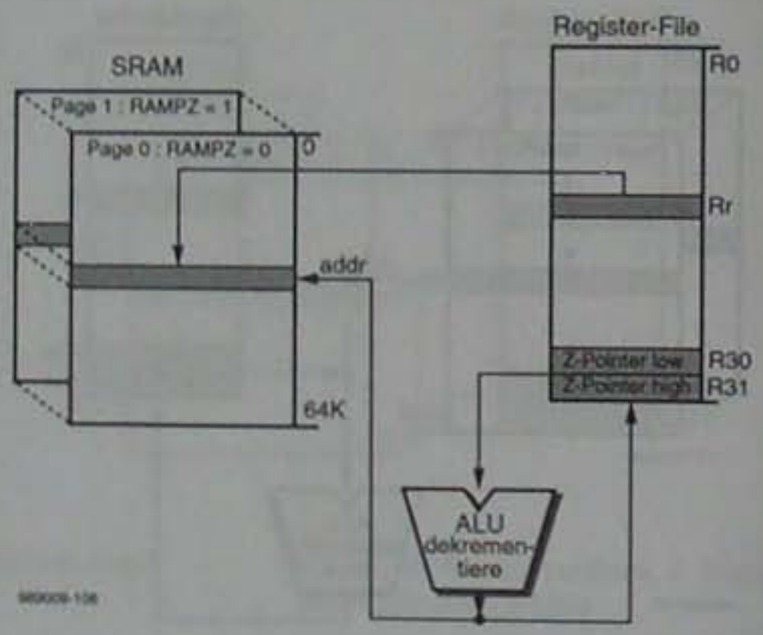
ST -Z,Rr

ST -Z,Rr — Speichere Register Rr indirekt über den um eins dekrementierten Z-Pointer.

Funktion: Z = Z − 1; anschließend (Z) ← Rr

Beschreibung: Speichert den Inhalt des Registers Rr in ein Byte des SRAM, welches durch den Wert des Z-Pointers (R31:R30) adressiert wird. Anschließend wird der Z-Pointer inkrementiert. Der Befehl ist für alle Register R0 bis R31 im Register-File zulässig. Der Zugriff auf das SRAM ist auf die selektierte 64KByte-Page beschränkt. (Nicht im AT90S1200 implementiert.)

Datenfluß:



Befehlsablauf: 1 Wort, 2 Maschinenzyklen, 2 Taktimpulse: 250 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

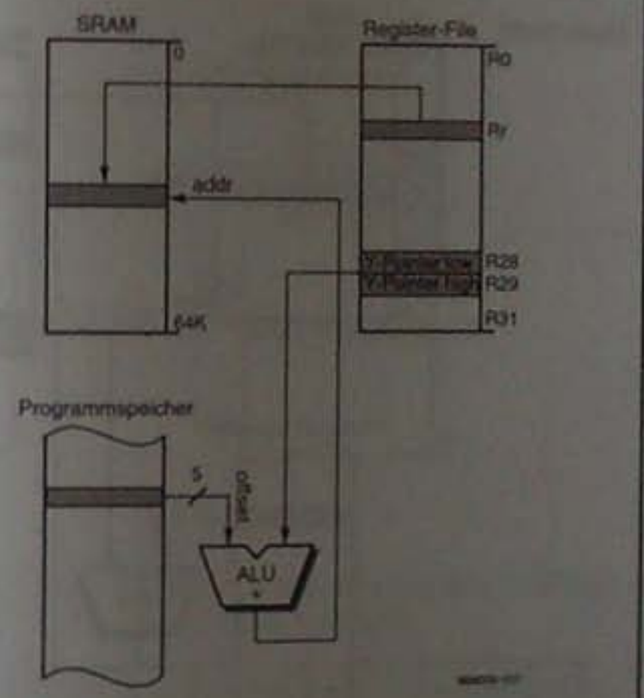
STD Y + offset,Rr

STD Y+ offset,Rr — Speichere Register Rr indirekt über Y-Pointer mit Offset

Funktion: (Y + offset) ← Rr

Beschreibung: Speichert den Inhalt des Registers Rr in ein Byte des SRAM, welches durch die Summe aus dem Wert des Y-Pointers (R29:R28) und dem Wert offset adressiert wird. Der Befehl ist für alle Register R0 bis R31 im Register-File zulässig. Der Zugriff auf das SRAM ist auf die selektierte 64KByte-Page beschränkt. (Nicht im AT90S1200 implementiert.)

Datenfluß:



Befehlsablauf: 1 Wort, 2 Maschinenzyklen, 2 Taktimpulse: 250 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

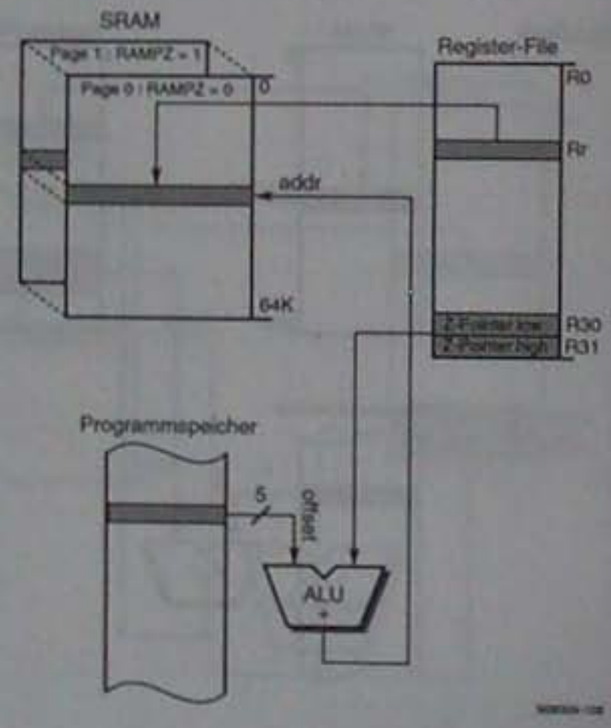
STD Z + offset,Rr

STD Z+ offset,Rr — Speichere Register Rr indirekt über Z-Pointer mit Offset.

Funktion: (Z + offset) ← Rr

Beschreibung: Speichert den Inhalt des Registers Rr in ein Byte des SRAM, welches durch die Summe aus dem Wert des Z-Pointers (R31:R30) und dem Wert offset adressiert wird. Der Befehl ist für alle Register R0 bis R31 im Register-File zulässig. Der Zugriff auf das SRAM ist auf die selektierte 64KByte-Page beschränkt. (Nicht im AT90S1200 implementiert.)

Datenfluß:



Befehlsablauf: 1 Wort, 2 Maschinenzyklen, 2 Taktimpulse: 250 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

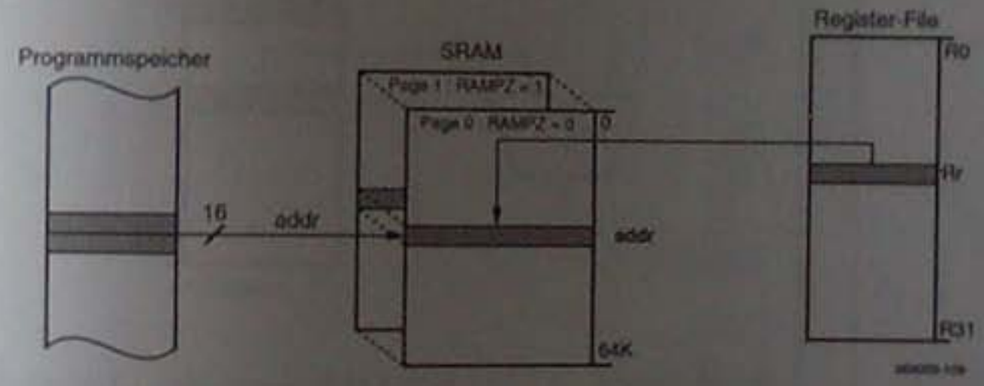
STS addr,Rr

STS addr,Rr — Speichere Register Rr direkt in das SRAM.

Funktion: SRAM(addr) ← Rr

Beschreibung: Speichert den Inhalt des Registers Rr direkt in ein Byte des SRAM, das über die Adresse addr (16 Bit) adressiert wird. Der Befehl ist für alle Register R0 bis R31 im Register-File zulässig. Der Wert addr darf im Bereich von 0 bis 65535 (dezimal) liegen. Der Zugriff auf das SRAM ist auf die selektierte 64KByte-Page beschränkt. (Nicht im AT90S1200 implementiert.)

Datenfluß:



Befehlsablauf: 2 Worte, 3 Maschinenzyklen, 3 Taktimpulse: 375 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

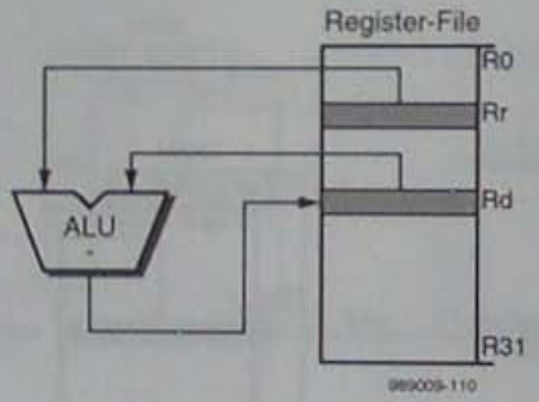
SUB Rd,Rr

SUB Rd,Rr — Subtrahiere Register Rr vom Register Rd

Funktion: Rd ← Rd – Rr

Beschreibung: Der Inhalt des Registers Rr wird vom Inhalt des Registers Rd subtrahiert. Das Ergebnis der Subtraktion steht im Register Rd. Der Inhalt des Registers Rr bleibt unverändert. Der Befehl ist für alle Register R0 bis R31 im Register-File zulässig.

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus, 1 Taktimpuls: 125 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

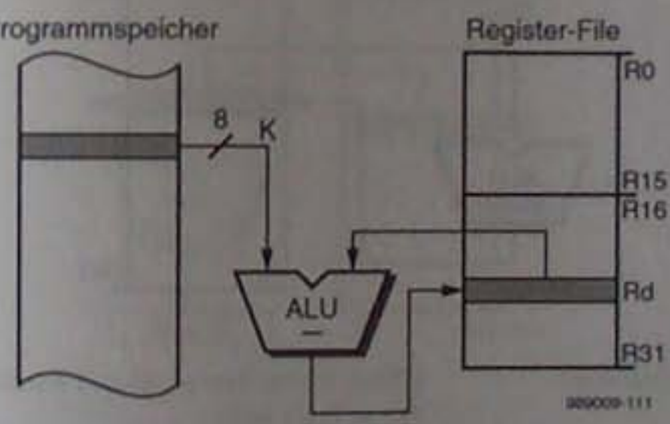
SUBI Rd,K

SUBI Rd,K — Subtrahiere unmittelbaren Wert K vom Register Rd.

Funktion: Rd ← Rd – K

Beschreibung: Der unmittelbare Wert K wird vom Inhalt des Registers Rd subtrahiert. Das Ergebnis der Subtraktion steht im Register Rd. Der Befehl ist die Register R16 bis R31 in der oberen Hälfte des Register-Files zulässig. Der unmittelbare Wert K kann im Bereich von 0 bis 255 (dezimal) liegen.

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus, 1 Taktimpuls: 125 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

SWAP Rd

SWAP Rd — Vertausche Nibbles des Registers Rd.

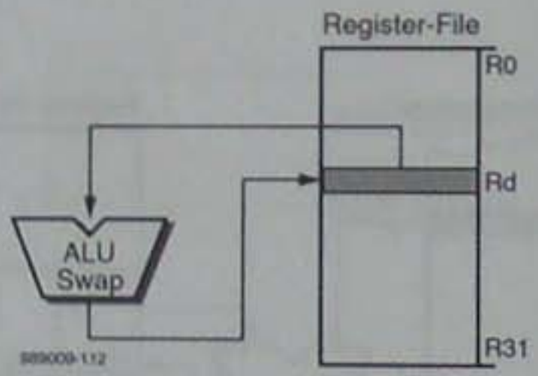
Funktion:

Rd<7:4> ← Rd<3:0>;
Rd<3:0> ← Rd<7:4>

Beschreibung:

Das höcherwertige und niederwertige Nibble (Halb-Byte) des Registers Rd werden vertauscht. Der Befehl ist für alle Register R0 bis R31 im Register-File zulässig.

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus, 1 Taktimpuls: 125 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

TST Rd

TST Rd — Teste Register Rd ob null oder negative.

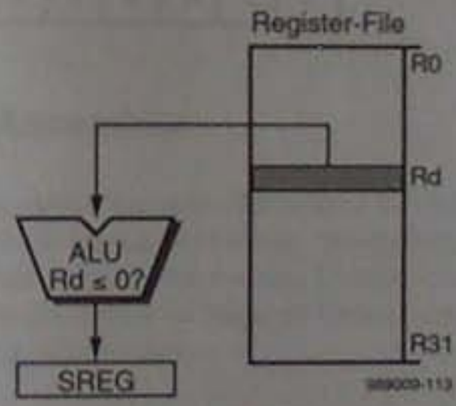
Funktion:

Rd ← Rd .AND. Rd

Beschreibung:

Der Inhalt des Registers Rd wird darauf überprüft, ob dieser null oder negativ ist. Dazu wird der Inhalt des Registers Rd mit sich selbst UND-verknüft. Der Befehl ist für alle Register R0 bis R31 im Register-File zulässig.

Datenfluß:



Befehlsablauf: 1 Wort, 1 Maschinenzyklus, 1 Taktimpuls: 125 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

WDR

WDR	Setze Watchdog-Timer zurück.
Funktion:	WDT ← 0
Beschreibung:	Der Watchdog-Timer wird zurückgesetzt.
Datenfluß:	
Befehlsablauf:	1 Wort, 1 Maschinenzyklus, 1 Taktimpuls: 125 ns bei 8 MHz

Flags:

I	T	H	S	V	N	Z	C

3 Software-Entwicklungstools

3.1 AVR-Assembler

Der AVR-Assembler übersetzt Mnemonics in 16-Bit Hex-Befehle für die AVR-Mikrocontroller-Familie. Sollten beim Assemblieren keine Fehler auftreten werden mehrere Dateien erzeugt, andernfalls wird eine Liste der Fehler im Message-Fenster ausgegeben und die Assemblierung abgebrochen.

Neben den Hex-Befehlen kann der Assembler auch ein Listing-File und ein Objekt-File erzeugen. Ferner kann der AVR-Assembler zusätzlich noch ein EEPROM-File erzeugen, das direkt in das EEPROM der AVR-Mikrocontroller programmiert werden kann. Das Hex-File dient zur späteren Programmierung des Mikrocontrollerprogrammspeichers (Flash). Das Objekt-File wird zur weiteren Verarbeitung mit dem Programm AVR-Studio (siehe Abschnitt 3.3) benötigt. Das Listing-File ist im Prinzip das ursprüngliche Programm-File, in dem Zeilennummer, ausgewertete Ausdrücke, Daten und Sourcecode stehen.

Der AVR-Assembler wird in zwei Versionen ausgeliefert: Die Windows-Version, die unter Microsoft Windows 3.11, Windows95 und

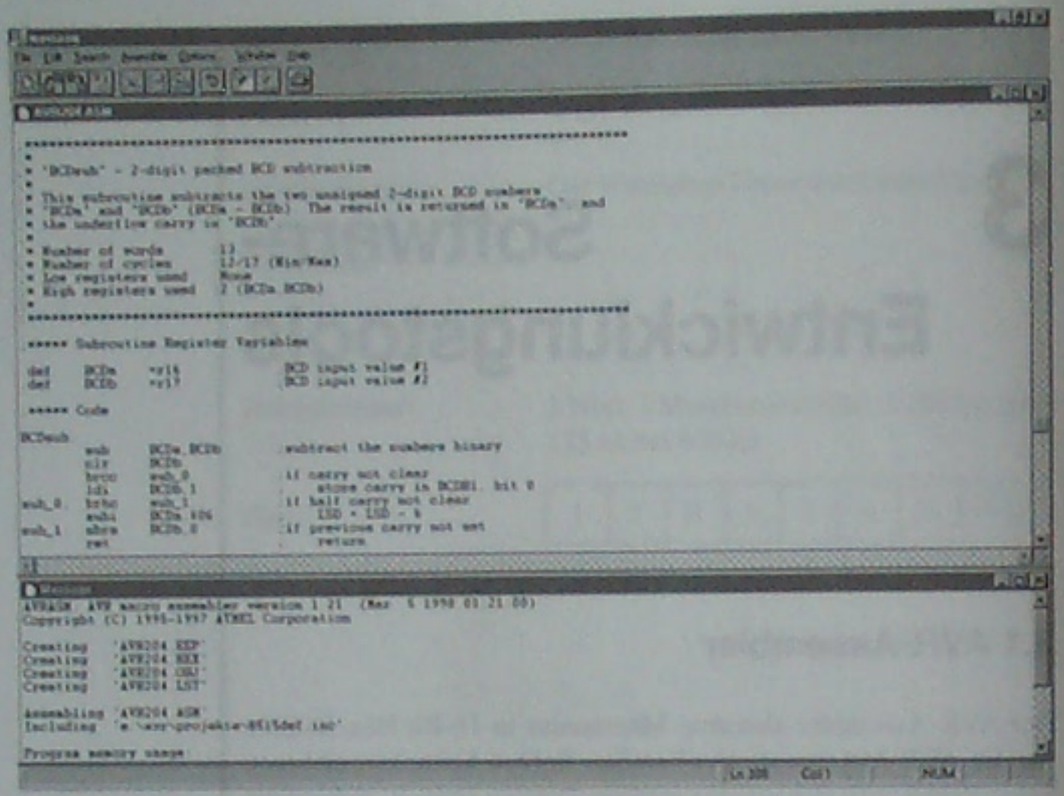


Bild 3.1: AVR-Assembler (Windows-Version).

WIndows NT (**Bild 3.1**), läuft und einer DOS-Version. Beide Versionen befinden sich auf der beigefügten CD (siehe Anhang A.1).

TIP: Setzen Sie im Menü „Options" die „Output-file extension" von „ROM" auf „HEX" um. Das erleichtert hinterher das laden der Hex-Datei mit dem Programm AVRProg. Dieses Programm erwartet nämlich die Endung „HEX" für diese Datei.

3.1.1 Direktiven

Was der Assembler ausführen soll, oder was bei der Ausführung zu beachten ist wird ihm in Direktiven mitgeteilt. Direktiven sind z. B. notwendig, um die Anfangsadresse des Objektcodes im AVR-Programmspeicher anzugeben oder, um andere Dateien einzubinden. In solchen INCLUDE-Dateien können bereits vordefinierte Ausdrücke, die man immer wieder benötigt, in den Sourcecode eingebunden werden. Die vom AVR-Assembler unterstützen Direktiven und deren Bedeutung sind in **Tabelle 3.1** zusammengefaßt.

Tabelle 3.1: Direktiven, die vom AVR-Assembler unterstützt werden.

Direktive	Erläuterung	Syntax
BYTE	Reserviert ein Byte für eine Variable odere mehrere Bytes für eine Tabelle	*Label:* .BYTE *Wert*
CSEG	Markiert den Beginn eines Code-Segments. In einem Code-Segmet steht das Programm oder man kann Werte in den Programmspeicher schreiben (vgl. DSEG).	.CSEG
DB	Definiert ein konstantes Byte oder mehrere konstante Bytes im Programmspeicher oder EEPROM, je nachdem ob dieseDirektive im Code- oder EEPROM-Segmet steht.	*Label:* .DB *Wert (,Wert...)*
DEF	Weist einem Symbol ein Register zu.	.DEF *Symbol = Register*
DEVICE	Teilt dem Assembler mit, für welchen AVR-Mikrocontrollertyp Sourcecode assembliert werden soll. Wird die DEVICE-Direktive verwendet, so warnt der Assembler, falls für den AVR-Typen nicht spezifizierte Anweisungen ausgeführt werden sollen.	Beispiel: .DEVICE AT90S2313
DSEG	Markiert den Beginn eines Datensegments. In einem Datensegment stehen üblicherweise nur Byte-Direktiven und Labels.	.DSEG
DW	Definiert ein konstantes 16-Bit Wort oder mehrere konstante 16-Bit Worte im Programmspeicher oder EEPROM, je nachdem ob diese Direktive im Code- oder EEPROM-Segmet steht.	*Label:* .DW *Wert (,Wert...)*

Direktive	Erläuterung	Syntax
EQU	Weist einem Symbol einen Wert zu. Das Symbol kann später nicht mehr umdefiniert werden.	.EQU *Symbol = Wert*
ESEG	Markiert den Beginn eines EEPROM-Segments. Im Gegensatz zum Datensegment, darf im EEPROM-Segment keine Byte-Direktive stehen.	.ESEG
EXIT	Teilt dem Assembler mit, die Assemblierung zu stoppen.	.EXIT
INCLUDE	Dient zur Einbindung anderer Dateien während der Assemblierung. Die INCLUDE-Direktive darf verschachtelt werden, d. h. eine Datei, die durch eine INCLUDE-Direktive in einen Sourcecode eingebunden wird, darf ihrerseits weitere INCLUDE-Direktiven beinhalten.	.INCLUDE *„Dateiname"* Falls nötig, muß im Dateinamen auch der Pfad angegeben werden.
LIST	Veranlaßt den Assembler, ein Listing-File zu erzeugen.	.LIST
LISTMAC	Veranlaßt den Assembler bei Auftreten eines Macros, das Macro an der aufrufenden Stelle auszuschreiben.	.LISTMAC
MACRO ENDMACRO	Definieren ein Makroanfang und -ende. Man kann über @0 bis @9 dem Makro zehn Parameter übergeben.	.MACRO *Name...* .ENDMACRO
NOLIST	Veranlaßt den Assembler, kein Listing-File zu erzeugen.	.NOLIST
ORG	Setzt den Anfang eines Code-, Daten oder EEPROM-Segments auf einen neuen Wert.	.ORG *Wert*
SET	Weist einem Symbol einen Wert zu. Das Symbol kann später umdefiniert werden.	.SET *Symbol = Wert*

3.1.2 Kommentare

Im Sourcecode können an beliebiger Stelle Kommentare stehen. Kommentare werden vom Assembler ignoriert und dienen nur zu Dokumentationszwecken. Ein Semikolon muß dem Kommentartext

vorangestellt werden. Zur besseren Lesbarkeit des Sourcecodes können auch Leerzeilen eingefügt werden. Leerzeilen brauchen nicht durch ein Semikolon kenntlich gemacht werden. Dazu einige Beispiele:

```
;Das ist ein Test
;
.equ    Zahl = 0x08     ;Dem Symbol Zahl wird der Wert
                        ;8 hex zugewiesen

        ldi   r16,0x10h ;Lade Register r16 mit 10 hex
```

3.1.3 Ausdrücke

Werte, die Symbolen zugewiesen werden sollen, können auch mathematische Ausdrücke sein. Diese Ausdrücke werden dann bei der Assemblierung ausgewertet. Der AVR-Assembler erlaubt folgende Ausdrücke:

Ausdruck	Bedeutung
!	Logisch Nicht
~	Bitweise Nicht
*	Multiplikation
/	Division
+	Addition
-	1. Subtraktion 2. Arithmethische Negation (Vorzeichenwechsel)
<<	Linksschieben
>>	Rechtsschieben
<	Kleiner
>	Größer
<=	Kleiner oder gleich
>=	Größer oder gleich
==	Gleich

Ausdruck	Bedeutung
!=	Ungleich
&	Bitweise Und-Verknüpfung
^	Bitweise Exklusiv-Oder-Verknüpfung
|	Bitweise Oder-Verknüpfung
&&	Logische UND-Verknüpfung
||	Logische ODER-Verknüpfung

Alle Ausdrücke können auch innerhalb von Klammern stehen. Ausdrücke innerhalb von Klammern werden erst ausgewertet, bevor diese weiter verknüpft werden.

3.1.4 Symbole und Marken

Der AVR-Assembler bietet die Möglichkeit bei der Programmierung statt mit absoluten Adressen, mit Symbolen und Marken zu arbeiten. Das erleichtert die Programmierung erheblich, da man sonst an jede Befehlszeile die aktuelle Adresse schreiben müßte, damit man immer weiß, wo man sich gerade befindet. Ferner müßte man im voraus wissen, wo sich die Unterroutine im Programmspeicher befinden wird, zu der verzweigt werden soll, bevor man diese überhaupt geschrieben hat. Alle diese Überlegungen werden mit der Einführung von Marken überflüssig. Jeder Routine, oder besser jeder Startadresse, an der sich eine Routine befindet, wird eine Marke zugewiesen. Möchte man eine Routine ausführen, wird zur entsprechenden Marke verzweigt. Bei der Assemblierung wird im ersten Durchgang den Marken die aktuelle Adresse zugewiesen. Der Assembler führt sozusagen Buch über die Adressen und der Programmierer braucht sich darüber keine Gedanken mehr zu machen.

Beispiel:

```
        clr   zaehler   ;loesche Variable zaehler
schleife: inc   zaehler   ;erhoehe zaehler um eins
        rjmp  schleife  ;wiederhole diesen Vorgang
```

In diesem Beispiel wird der Marke „schleife:" die Adresse zugeordnet, an der der Befehl zum Inkrementieren „inc zaehler" steht. Anschließend kann zu dieser Marke mit dem Befehl „rjmp" gesprungen werden. Ohne Marken müßte nach dem Sprungbefehl die absolute Adresse stehen, an der sich der INC-Befehl befindet. Ein weiterer Nachteil wäre, daß dann das Programm nicht innerhalb des Speichers verschoben werden könnte, da sich dann die absolute Adresse ändern würde.

Marken müssen mit einem Doppelpunkt „:" enden. Ferner dürfen sie mit gleichen Namen im Programm nur einmal vergeben werden, da der AVR-Assembler ansonsten die Assemblierung mit einer Fehlermeldung abbricht.

Das Arbeiten mit unmittelbaren oder auch direkten Werten läßt sich genauso rationalisieren. Wird z. B. im Programm an einer an oder mehreren Stellen ein bestimmter Wert gebraucht, so kann dieser vorher einem Symbol zugeordnet werden. An der entsprechenden Programmstelle schreibt man dann den Symbolnamen statt des Wertes. Diese Vorgehensweise trägt wesentlich zur besseren Lesbarkeit von Programmen bei, da man den Werten sinnvolle Symbole zuordnet kann.

Beispiel:

```
.equ    zahl1 = 4    ;Symbol „zahl1" wird eine
                     ;4 zugeordnet
.equ    zahl2 = 7    ;Symbol „zahl2" wird eine
                     ;7 zugeordnet
```

3.1.5 Datentypen

Der AVR-Assembler verarbeitet binäre, dezimale, hexadezimale und ASCII-Werte. In der folgenden Tabelle sind die unterstützten Datentypen mit Syntax und je einem Beispiel zusammengefaßt.

Typ	Beispiel
binär	0b10100110 (binärer Wert)
dezimal	100 (dezimaler Wert 100)
hexadezimal	0x30 oder $30 (hexadezimaler Wert 30hex)
ASCII	1. 'A' (ASCII-Zeichen A) bei Befehlen 2. "A" bei der DB-Direktive

Bei der DB-Direktive können die Zeichen als eine ASCII-Kette geschrieben werden. Das ist bei zum Beispiel in Tabellen sehr hilfreich.

Beispiel:

```
Tabelle: .db "Hallo"
```


3.1.6 Funktionen

Der AVR-Assembler kennt folgende Funktionen:

Funktion	Erläuterung	Syntax
LOW	Liefert das niederwertige Byte eines Wertes.	LOW(*Wert*)
HIGH	Liefert das höherwertige Byte eines Wertes.	HIGH(*Wert*)
BYTE2	Liefert das höherwertige Byte eines Wertes (vgl. HIGH).	BYTE2(*Wert*)
BYTE3	Liefert das dritte Byte eines Wertes.	BYTE3(*Wert*)
BYTE4	Liefert das vierte Byte eines Wertes.	BYTE4(*Wert*)
LWRD	Liefert die Bits 0 bis 15 eines Wertes (Low Word).	LWRD(*Wert*)
HWRD	Liefert die Bits 16 bis 31 eines Wertes (High Word).	HWRD(*Wert*)
PAGE	Liefert die Bits 16 bis 21 eines Wertes.	PAGE(*Wert*)
EXP2	Berechnet die Zweierpotenz.	EXP2(*Wert*)
LOG2	Berechnet den ganzzahligen Teil des Logarithmus zur Basis 2	LOG2(*Wert*)

3.2 AVR-Simulator

Der AVR-Simulator von Atmel wird nicht länger unterstützt und ist bereits abgekündigt. Die Aufgaben des AVR-Simulators werden bestens von dem Programm AVR-Studio (siehe Abschnitt 3.3) ausgeführt, so daß es keinen Sinn mehr macht, zwei Programme parallel nebeneinder zu haben. Auf die Beschreibung dieses veralteten Softwaretools wird deshalb an dieser Stelle verzichtet. Im nächsten Abschnitt wird auf das aktuelle Simulationstool AVR-Studio eingegangen werden.

3.3 AVR-Studio

Mit dem Programm AVR-Studio kann man alle AVR-Mikrocontroller simulieren. Dazu muß zunächst mit dem AVR-Assembler aus dem Sourcecode ein Object-File erzeugt werden. AVR-Studio ist dann in der Lage, dieses Object-File zu lesen und die Programmausführung zu simulieren. Ist ein In-Circuit-Emulator am PC angeschlossen, dann dient AVR-Studio als Emulatorsteuerung (siehe Kapitel 4).

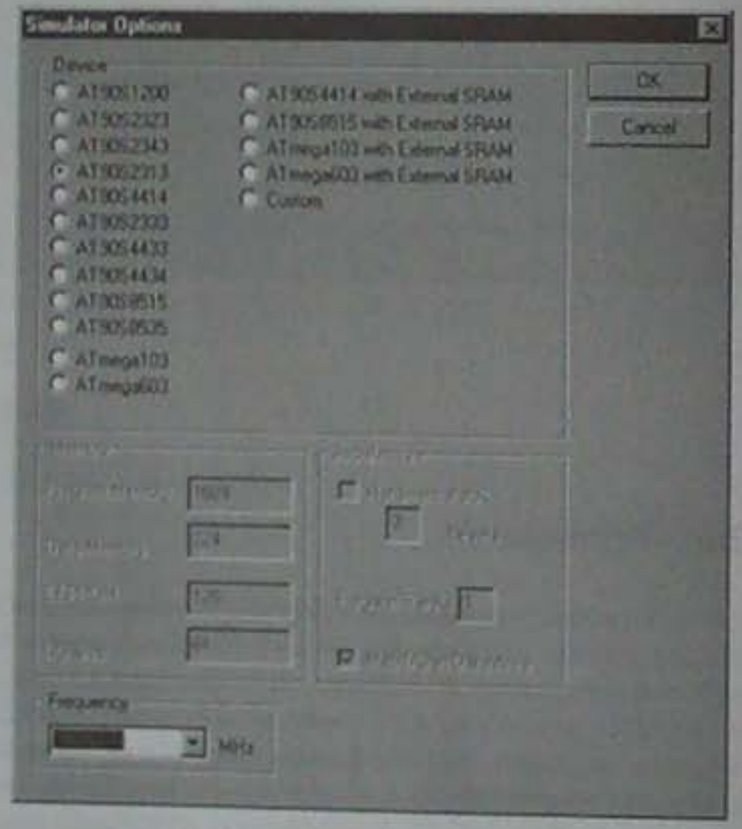


Bild 3.2: Das Fenster „Simulator Options".

AVR-Studio befindet sich auf der Begleit-CD im Verzeichnis \Bin. Das Programm ist nur unter Windows95 und Windows NT lauffähig.

Wird mit AVR-Studio das erste Mal ein bestimmtes Object-File geöffnet, so erscheint das „Simulator Options" Fenster. In diesem Fenster muß man den zu simulierenden AVR-Mikrocontrollertyp und die zu verwendende Taktfrequenz angeben (**Bild 3.2**).

Hat man die entsprechende Wahl getroffen, erscheint das Arbeitsfeld des AVR-Studios. In diesem Feld kann man verschiedene Fenster öffnen, die z. B. Informationen über den Prozessor geben oder den Inhalt der Register anzeigen. **Bild 3.3** zeigt ein typisches Aussehen des AVR-Stuido Arbeitsfeldes mit geöffneten Fenstern.

Bild 3.3: AVR-Studio im Simulator-Modus.

Im AVR-Studio lassen sich folgende Fenster öffnen:

Fenstername	Bedeutung
Source Window	Der Sourcecode wird angezeigt.
Watch Window	Zeigt die Variablen, Variablentypen und deren Werte an. Der AVR-Assembler erzeugt keine symbolische Information, so daß dieses Fenster nur bei Verwendung eines C-Compilers genutzt werden kann.
Register Window	Zeigt den Inhalt der 32 Register im Register-File an.
Message Window	Zeigt Nachrichten von AVR-Studio an den Benutzer an.
Memory Window	Zeigt den Inhalt der unterschiedlichen Speicher an. Der Inhalt kann in diesem Fenster auch modifiziert werden.
Processor Window	Zeigt den Programmzähler, den Stack-Pointer, den Maschinenzyklus-Zähler und die Flags des Status-Registers an.
Peripheral Device Windows	Hier können Fenster, die Informationen für die verschiedenen Peripherie-Module wie Timer/Counter, Ports, EEPROM-Register, SPI und UART anzeigen, geöffnet werden.

Die AVR-Studio Befehle wie Einzelschritt, Start und Stop können mit der Maus durch Anklicken der Icons ausgeführt werden. Folgende Befehle sind auch durch Tastenkombinationen erreichbar:

Befehl	Tastenkombination
Toggle Register Window	Alt+0
Toggle Watch Window	Alt+1
Toggle Message Window	Alt+2
Toggle Processor Window	Alt+3
Add Memory Window	Alt+4
Show Breakpoint List	Ctrl+B
Copy to Clipboard	Ctrl+C
Open File	Ctrl+O
Help	F1
Run	F5
Break	Ctrl+F5
Reset	Shift+F5
Run to Cursor	F7
Toggle Breakpoint	F9
Step Over	F10
Trace Into	F11
Step Out	Shift+F11

4 Hardware-Entwicklungstools

In diesem Kapitel werden drei Hardware-Entwicklungstools besprochen: Der AVR-Programmer, der AVR AT90ICEPRO In-Circuit-Emulator von Atmel und das Mixed-Signal-Oszilloskop mit 16-Kanal Logikanalysator HP54645D von Hewlett Packard.

Mit dem AVR-Programmer kann man alle AVR-Mikrocontroller (außer die ATmega-Familie) seriell oder auch In-Circuit programmieren. Dabei handelt es sich um ein sehr preiswertes Tool, so daß der interessierte Leser einen leichten Einstieg in die Welt der AVR-Mikrocontroller erhält. Die beiden anderen Tools, Emulator und Logikanalysator sind für die professionelle Hard- und Softwareentwicklung gedacht. Preislich liegt der Emulator in der Größenordung von 4.000 DM und der Logikanalysator kostet ca. 9.500 DM, so daß beide Geräte das Budget eines Hobbyisten wohl sprengen würden. Die Bezugsadressen und Kontaktmöglichkeiten für die beschriebenen Tools sind im Anhang angegeben.

4.1 AVR-Programmer

4.1.1 Hardware des AVR-Programmers

Hat man ein Programm erfolgreich geschrieben und simuliert, steht man vor der Aufgabe, dieses in einen AVR zu brennen. Ein preiswertes Gerät dazu ist der AVR-Programmer, dessen Schaltplan in **Bild 4.1a** und **Bild 4.1b** zu sehen ist.

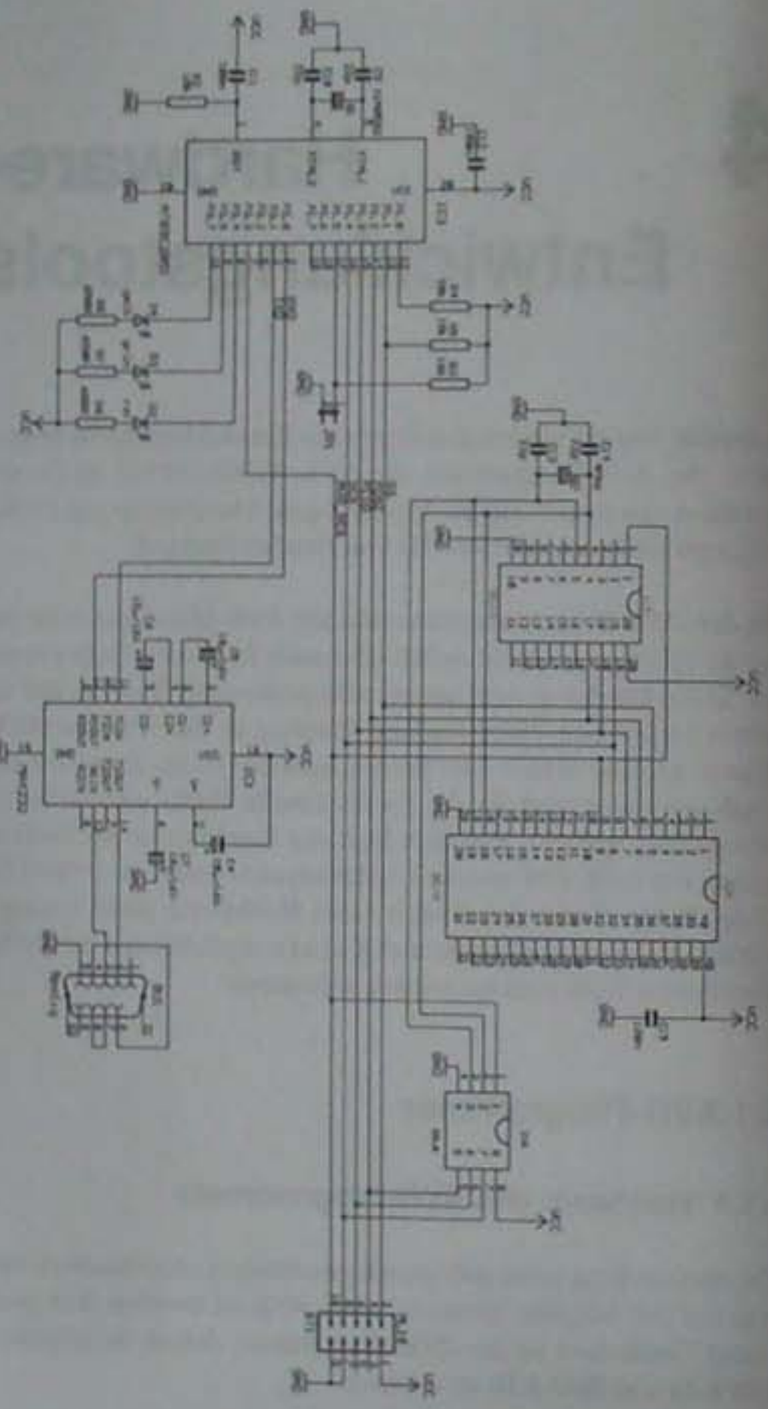


Bild 4.1a: Schaltplan des AVR-Programmers.

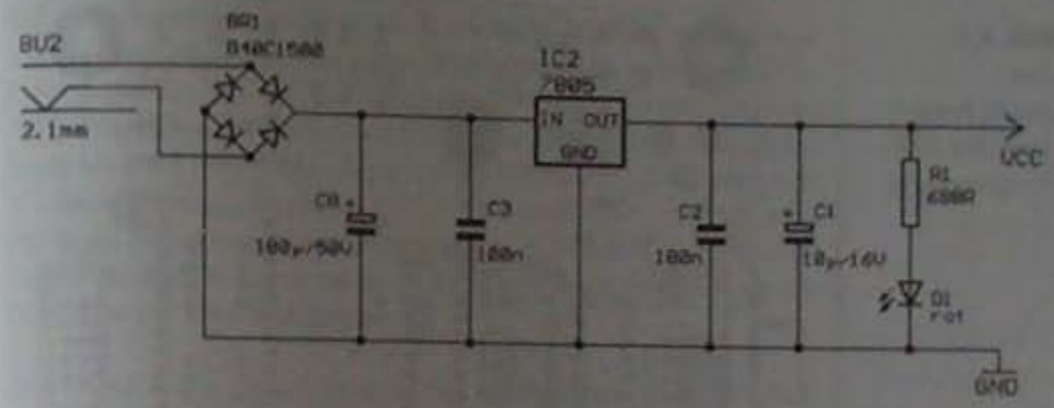


Bild 4.1b: Schaltplan des Netzteils zum AVR-Programmer.

Kernstück des AVR-Programmers ist ein programmierter Mikrocontroller AT89C2051 der Firma Atmel. Dieser übernimmt die Kommunikation mit einem PC, auf dem die Programmiersoftware läuft, über die serielle RS-232 Schnittstelle. Ferner erzeugt er die richtigen Signale für die serielle Programmierung der AVR-Mikrocontroller. Die Pegelwandlung zwischen serieller Schnittstelle des PCs und dem AVR-Programmer übernimmt ein MAX232 o. ä. Baustein. Programmiert werden können sowohl das Flash, als auch das interne EEPROM der AVR-Mikrocontroller. Die drei LEDs D2 bis D4 zeigen den Betriebszustand des AVR-Programmers an:

- Gelbe LED: AVR-Mikrocontrooler wird programmiert bzw. verglichen
- Grüne LED: Vergleich war erfolgreich
- Rote LED: Vergleich war nicht erfolgreich

Beim Einschalten des AVR-Programmers kommt diesen drei LEDs eine weitere Bedeutung zu: Der AVR-Programmer führt einen Selbsttest durch. Ist dieser erfolgreich, so leuchten alle drei LEDs nacheinander auf und bleiben an. Schließlich gehen alle drei LEDs gleichzeitig aus. Zur Programmierung der unterschiedlichen AVR-Mikrocontroller sind drei Fassungen, 8polig, 20polig und 40polig auf der Platine vorgesehen. Natürlich können nicht gleichzeitig mehrere oder unterschiedliche Typen programmiert werden. Möchte man einen AVR-Mikrocontroller In-Circuit programmieren, z. B. wie dies mit

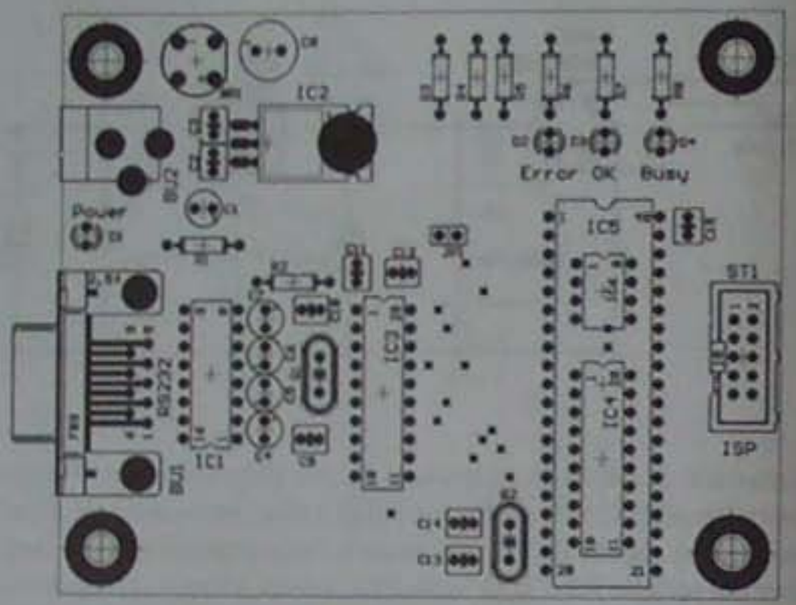


Bild 4.2: Der Bestückungsplan des AVR-Programmers.

Stückliste AVR-Programmer	
	D4 = LED gelb, 3 mm
Widerstände:	IC1 = MAX232 o .ä.
	IC2 = Festspannungregler 7805
R1, R6...R8 = 680 Ω	IC3 = Mikrocontroller AT89C2051
R2...R5 = 10 kΩ	
	Außerdem:
Kondensatoren:	
	Q1 = Quarz 11,0592 MHz
C1, C4.C7 = 10 µF/16V	Q2 = Quarz 4 MHz
C2, C3, C11, C12, C15 = 100 nF	DIL8 = IC-Fassung 8polig
C9, C10, C13, C14 = 22 pF	DIL20 = IC-Fassung 20polig
C8 = 100 µF/50V	DIL40 = IC-Fassung 40polig
	JP1 = nicht bestücken
Halbleiter:	ST1 = Wannenstecker 2 x 5polig
	BU1 = Sub-D-Buchse 9polig
BR1 = Brückengleichrichter B40C1500	BU2 = Niederspannungsbuchse Stift 2,1mm
D1, D2 = LED rot, 3 mm	Kabel für serielle Schnittstelle, 1:1 durch-
D3 = LED grün, 3 mm	verbunden (kein Nullmodem-Kabel!)

Tabelle 4.1: Stückliste des AVR-Programmers mit Netzteil.

der AVR-Experimentierplatine in Kapitel 5 möglich ist, so kann eine Verbindung über den 10poligen Wannenstecker zwischen AVR-Programmer und Zielhardware hergestellt werden. Versorgt wird der AVR-Programmer über das Netzteil, das in **Bild 4.1b** abgebildet ist. Dazu wird noch eine Gleichspannung zwischen 9...20 V oder eine Wechselspannung zwischen 6,5 V...15 V benötigt. Die LED D1 zeigt an, ob der AVR-Programmer eingeschaltet ist. Die AVR-Programmerplatine ist in **Bild 4.2** zu sehen. Auf ihr sind beide Schaltungen aus **Bild 4.1a** und **Bild 4.1b** aufgebaut.

Auf der AVR-Programmerplatine sind standardmäßig nur DIL-Fassungen zur Aufnahme von AVR-Mikrocontrollern vorgesehen. Möchte man sogenannte Nulldruckfassungen einsetzen, so kann man ebenfalls über den 10poligen Wannenstecker eine Verbindung zu Apaptern (**Bild 4.3a/b** und **4.4a/b**) herstellen. Denkbar sind auch Adapter zur Aufnahme von Mikrocontrollern in PLCC-Gehäuse.

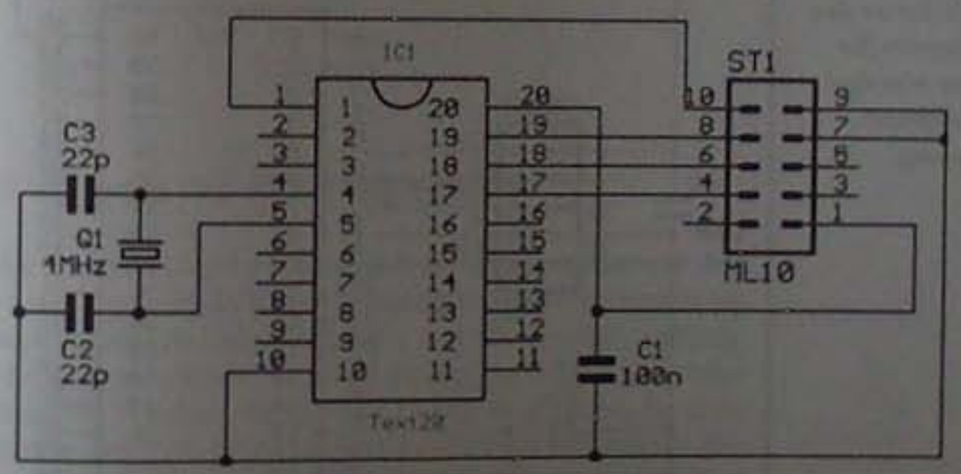


***Bild 4.3a:** Schaltplan des Adapters für eine 20polige Nulldruckfassung.*

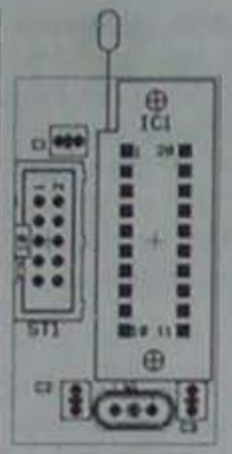


Bild 4.3b:
Platine des Adapters für eine 20polige Nulldruck-fassung.

Kondensatoren:

C1 = 100 nF
C2, C3 = 22 pF

Außerdem:

IC1 = Nulldruckfassung 20polig
Q1 = Quarz 4 MHz
ST1 = Wannenstecker 2 x 5polig

Tabelle 4.2:
Stückliste der 20poligen Adapterplatine.

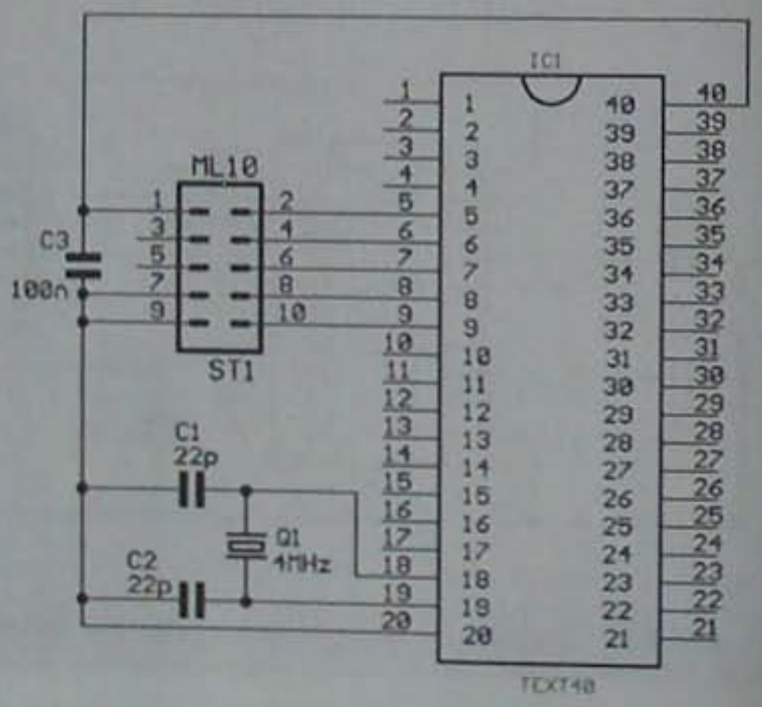
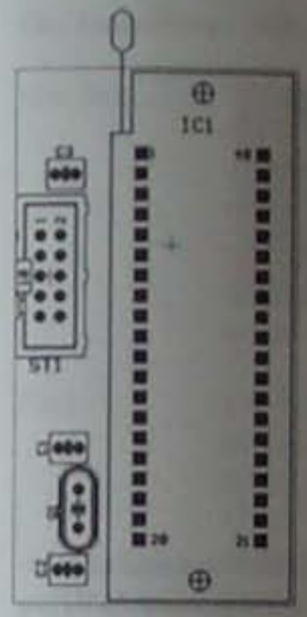


Bild 4.4a:
Schaltplan des Adapters für eine 40polige Nulldruck-fassung.



Kondensatoren:

C1, C2 = 22 pF
C3 = 100 nF

Außerdem:

IC1 = Nulldruckfassung 40polig
Q1 = Quarz 4 MHz
ST1 = Wannenstecker 2 x 5polig

Bild 4.4b:
Platine des Adapters für eine 40polige Nulldruck-fassung

Tabelle 4.3:
Stückliste der 40poligen Adapterplatine.

4.1.2 Software des AVR-Programmers

Für die Kommunikation zwischen dem AVR-Programmer und dem PC ist das Programm **AVRProg** zuständig. Dieses Programm ist unter den Betriebssystemen Windows95 und WindowsNT lauffähig. Das Programm befindet sich auf der beiliegenden CD. Neben AVRProg für Windows, enthält die CD auch Programme, die es ermöglichen, den AVR-Programmer unter MS-DOS zu betreiben. Bevor man das Programm AVRProg starten kann, muß es installiert werden. Nach erfolgreicher Installation kann das Programm, wie unter Windows üblich, gestartet werden. Sollte AVRProg den AVR-Programmer nicht finden, da dieser z. B. nicht an einem COM-Port angeschlossen und/oder nicht eingeschaltet ist, dann wird das von AVRProg gemeldet (siehe dazu **Bild 4.5**).

Bild 4.5: Das Programm AVRProg meldet, daß der AVR-Programmer nicht gefunden wurde.

Funktioniert der AVR-Programmer ordnungsgemäß und sind alle Verbindungen korrekt, so sollte nach dem Start des Programms AVRProg auf dem Bildschirm ein Fenster wie in **Bild 4.6** zu sehen sein.

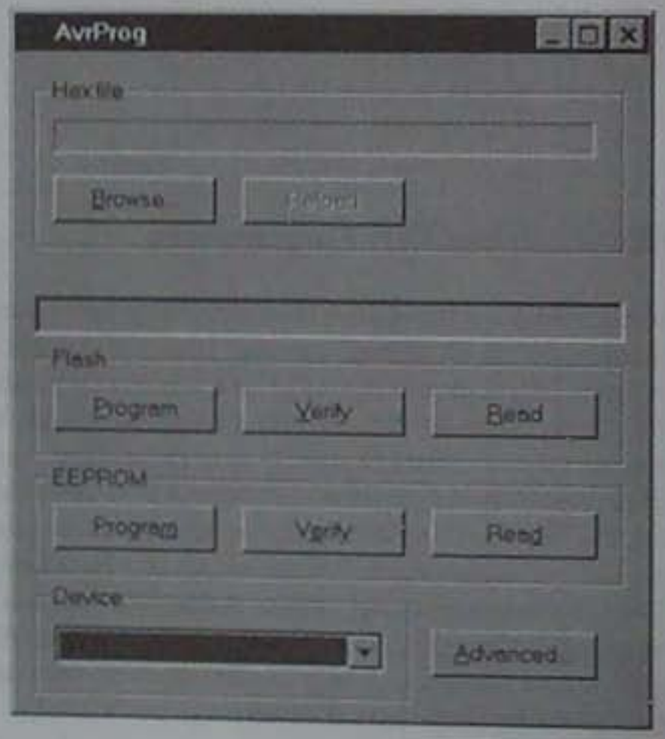


Bild 4.6: Das Hauptfenster des Programms AVRProg, wenn beim Start alles korrekt ist.

Das Hauptfenster (**Bild 4.6**) ist in vier Gruppen aufgeteilt:

Hex file: Hier kann eine Datei im Intel-Hex-Format in den Datei-Buffer von AVRProg geladen werden. Mit dem Button „Browse" kann man die Verzeichnisse nach einer solchen Datei durchblättern.

Mit „Reload" kann eine Datei, dessen Pfad bereits eingegeben ist, im Datei-Buffer von AVRProg aktualisiert werden.

Flash: Mit dem Button „Program" kann der Programmspeicher des aufgewählten AVR-Mikrocontrollers programmiert werden. Dabei werden die Daten, die sich im Datei-Buffer von AVRProg befinden, programmiert.

Mit „Verify" werden die Daten im Programmspeicher des ausgewählten AVR-Mikrocontrollers mit den Daten, die sich im Datei-Buffer von AVRProg befinden, verglichen.

Schließlich werden mit „Read" die Daten aus dem ausgewählten AVR-Mikrocontroller gelesen und in die Datei geschrieben, die im Feld „Hex file" angegeben ist.

EEPROM: Die Buttons dieser Gruppe haben die gleichen Funktionen, wie die der Gruppe „Flash" mit der Ausnahme, daß diese sich jeweils auf das interne EEPROM und nicht auf den Programmspeicher des AVR-Mikrocontrollers beziehen.

Device: Hier kann mit Hilfe eines Menüs der gewünschte AVR-Mikrocontroller-Typ eingestellt werden.

Ferner befindet sich in der unteren, rechten Ecke des Hauptfensters ein zusätzlicher Button mit der Aufschrift „Advanced...". Klickt man auf diesen Button, öffnet sich ein weiteres Fenster, in dem man einige zusätzliche Angaben machen kann (siehe **Bild 4.7**).

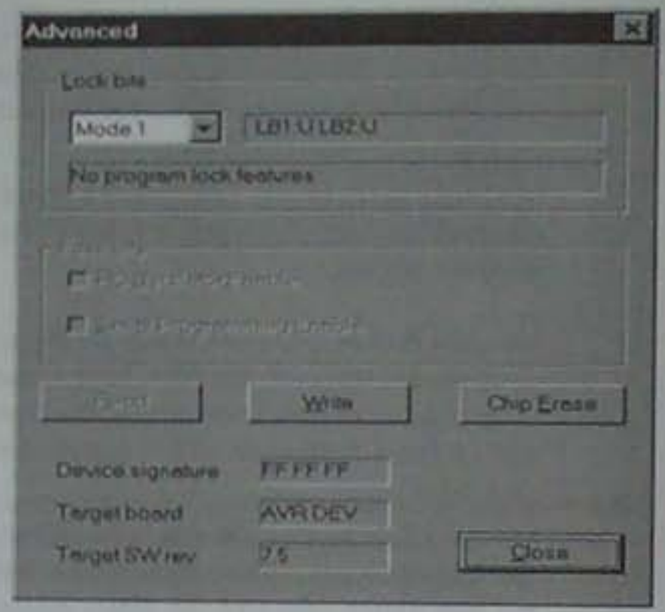


Bild 4.7: Das Fenster für die zusätzlichen Angaben („Advanced...").

Lock bits: Im „Lock bits" Menü kann man bestimmen, wie man diese Bits setzen möchte. Zur Ausführung muß man anschließend auf den „Write"-Button klicken. Unterhalb des Menüs wird zu jeder Auswahl aus dem Menü eine kurze Beschreibung gegeben. Es ist anzumerken, daß die Lock-Bits nur gelöscht werden können, indem man auf den Button „Chip Erase" klickt oder den Programmspeicher (Flash) des AVR-Mikrocontrollers programmiert. Die Lock-Bits können bei früheren Version der AVR-Mikrocontroller AT90S1200 Revision A-C im seriellen Programmiermodus nicht programmiert werden. Sollte im Hauptfenster einer dieser Typen ausgewählt worden sein, bleibt diese Gruppe unaktiviert.

Fuse bits: Die Gruppe bleibt unaktiviert, da diese Funktion nur im parallelen Programmiermodus der AVR-Mikrocontroller ausgeführt werden kann. Da der AVR-Programmer nur den seriellen Programmiermodus beherrscht, wird diese Funktion nicht unterstützt.

Chip Erase: Durch klicken auf diesen Button wird der Programmspeicher (Flash) und das EEPROM des AVR-Mikrocontrollers gelöscht. Ferner werden die Lock-Bits gelöscht. Die Fuse-Bits werden nicht beeinflußt.

Device signature: Zeigt die Signature-Bytes des AVR-Mikrocontrollers an. Im seriellen Programmiermodus können die Signature-Bytes nicht gelesen werden, wenn die Lock-Bits gesetzt sind.

Target board: Zeigt an, welche AVR-Programmer-Platine angeschlossen ist. Für den AVR-Programmer, der in diesem Buch beschrieben wird, wird immer „AVR DEV" angezeigt.

Target SW rev.: Zeigt die Version der Software an, mit der der Mikrocontroller AT89C2051 arbeitet, der sich auf der AVR-Programmer-Platine befindet.

Während des Betriebs des AVR-Programmers und des Programms AVRProg können sich noch andere Fenster öffnen, die aber selbsterklärend sind.

4.2 Der Emulator AVR AT90ICEPRO

Mit dem Emulator AVR AT90ICEPRO ist es u. a. möglich, Programme im Einzelschritt (Single-Step) abzuarbeiten, Haltepunkte (Breakpoints) im Programm zu setzen, an den die Programmausführung angehalten wird, den Inhalt der Register zu modifizieren und Programme in Echtzeit (Real-Time) ausführen zu lassen. Die Bedienung des Emulators erfolgt mit der Software AVR-Studio, der in Verbindung mit dem AVR AT90ICEPRO zur Emulatorsoftware wird. Findet AVR-Studio beim Laden einer Datei den Emulator AVR AT90ICEPRO an eine serielle Schnittstelle, schaltet AVR-Studio vom Simulations- in den Emulationsmodus um. Dies wird durch die unterste Zeile rechts angezeigt (**Bild 4.8**).

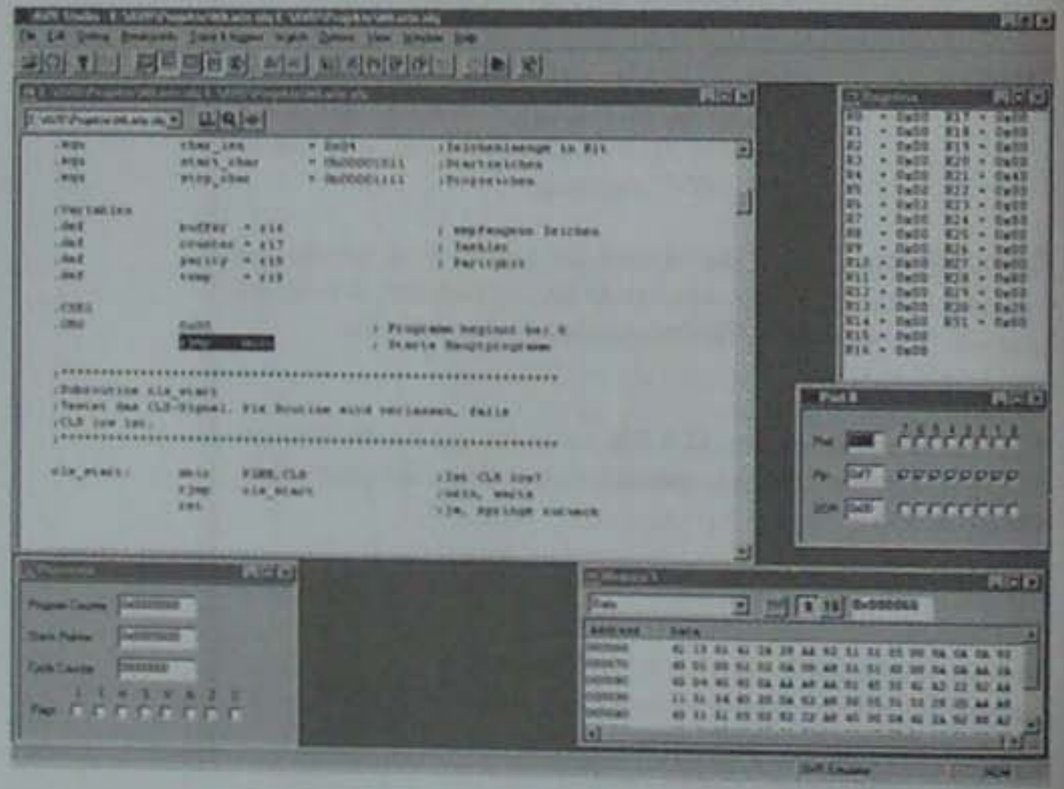


Bild 4.8: *AVR-Studio als Emulatorsoftware.*

Die Bedienung von AVR-Studio in Verbindung mit dem Emulator ist vollständig analog zur Beschreibung in Abschnitt 3.3 und wird deshalb hier nicht wiederholt.

Der Emulator AVR AT90ICEPRO unterstützt z. Z. (Stand 1/99) folgende AVR Mikrocontroller:

- AT90S1200
- AT90S2313
- AT90S2323
- AT90S2343
- AT90S4414
- AT90S8515

Ferner ist es möglich, durch Upgrade-Kits, zukünftige AVR-Mikrocontroller zu unterstützen. Zum Zeitpunkt der Drucklegung existiert bereits ein Upgrade-Kit für den AT90S8535.

4.3 Logikanalysator HP54645D

Bild 4.9: HP54645D Oszilloskop mit 16-Kanal Logikanalysator.

Der Logikanalysator HP54645D besteht aus einem 2-Kanal Oszilloskop mit einer Bandbreite von 100 MHz und einem 16-Kanal Logikanalysator. Das besondere an diesem Gerät ist, daß alle 18 Kanäle zeitkorreliert sind. Ferner können sowohl analoge Signale als auch digitale Signale zum Triggern herangezogen werden.

Bei der Entwicklung von Schaltungen mit Mikrocontrollern und/oder analogen Schaltungsteilen leistet dieses Gerät hervorragende Dienste. Als kleines Beispiel sei auf **Bild 5.34** in Abschnitt 5.6 verwiesen, das mit diesem Gerät aufgenommen worden ist.

5 Anwendungen

Im folgenden Kapitel werden mehrere Projekte vorgestellt, die sich sehr einfach und mit nur geringem zusätzlichen Aufwand an Bauelementen mit den AT90S2313 realisieren lassen. Die abgedruckten Programme lassen sich sofort in eigene Projekte einbinden, da sie modular aufgebaut sind. Es wurde großen Wert auf einfache Programmierung gelegt, damit der Code verständlich bleibt. Das jeweils dargestellte Ausführungsbeispiel ist sehr einfach ausgelegt. Das ermöglicht die Veranschaulichung der aufgerufenen Programmodule, ohne das gesamte Programm unnötig aufzublähen.

Auch eine Programmsammlung kommt ohne ein wenig Hardware nicht aus. Damit der Leser nicht eine ganze Sammlung an Experimentierplatinen und unterschiedlichen Mikrocontrollern auf Lager legen muß, basieren alle in diesem Buch besprochenen Schaltungen auf einer einzigen Experimentierplatine (**Bild 5.1a** und **Bild 5.1b**). Auf dieser Platine können die Mikrocontroller AT90S1200 und AT90S2313 eingesetzt werden. Die Platine kann vom Elektronik Laden Detmold (Anschrift siehe Anhang) bezogen werden.

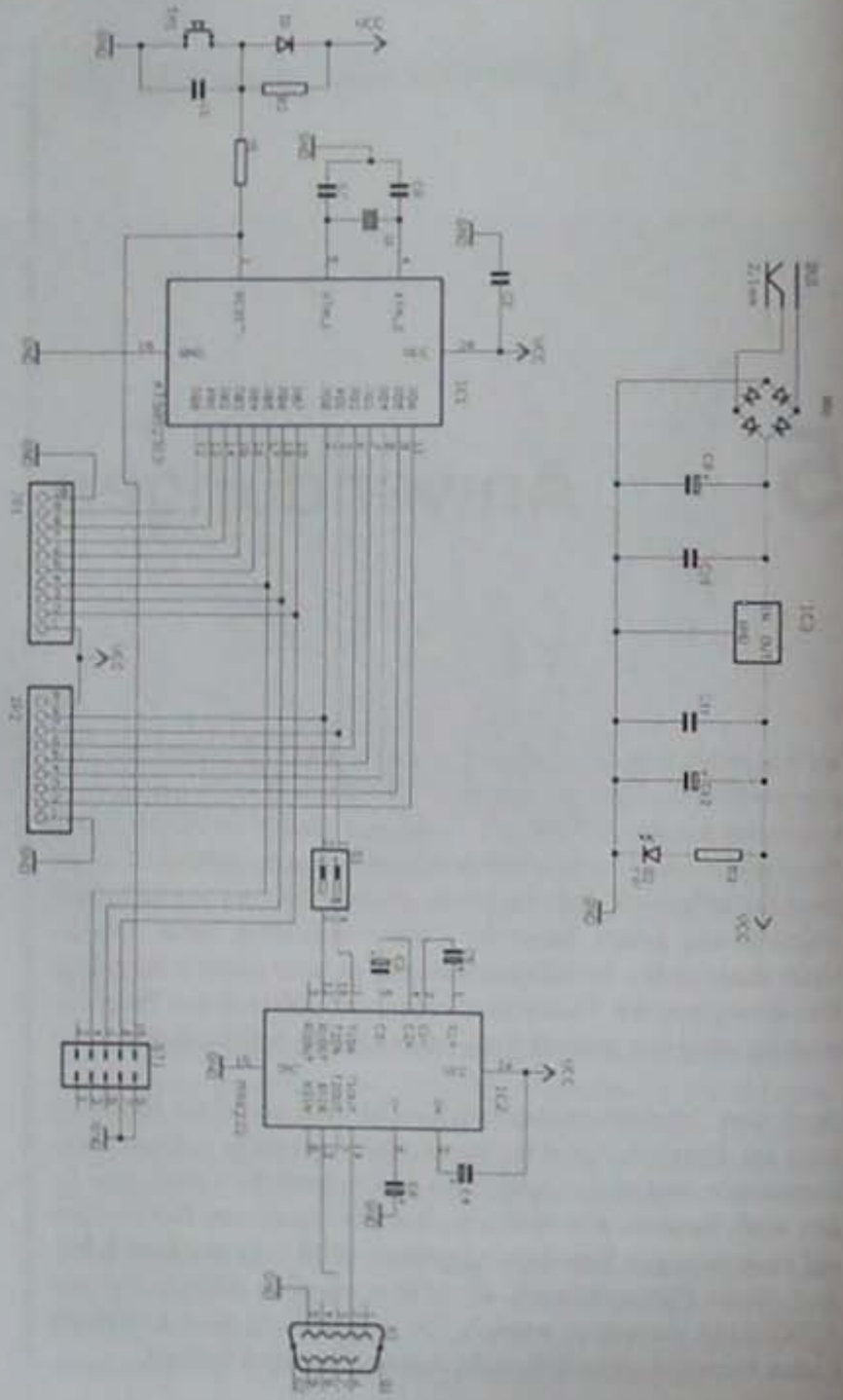


Bild 5.1a: Schaltplan der AVR-Experimentierplatine.

Der Mikrocontroller ist in seiner einfachsten Form beschaltet. Für den Takt sorgt der Quarz Q1 mit 4 MHz Resonanzfrequenz. Die Kondensatoren C7 und C8 sorgen dafür, daß der Quarz sicher anschwingt. Die RC-Kombination aus R2 und C1 verzögern das Reset-Signal am Pin „RESET" solange, bis die Versorgungsspannung stabil ist. Über den Taster TA1 läßt sich bei Bedarf manuel ein Reset auslösen. Der Widerstand R1 begrenzt den Strom, den der RESET-Eingang aufnimmt. Hochfrequente Störungen auf der Versorgungsspannung werden vom Kondensator C2 unterdrückt. Für Experimente mit der seriellen Schnittstelle RS232 ist ein Pegelwandler, er besteht aus IC2, den Kondensatoren C3 bis C6 und der Buchse X1, auf der Platine integriert. Möchte man diesen nicht an den Portleitungen des Mikrocontrollers hängen haben, so kann er über den Schalter S1 abgeklemmet werden. Die In-Circuit-Programmierung des Mikrocontrollers ist über den Stecker ST1 möglich. Damit läßt sich der AVR-Programmer aus Kapitel 4 direkt an die Experimentierplatine anschließen und der Mikrocontroller ohne Hin- und Herstecken programmieren. Die übrigen Bauelemente BU1, BR1, IC3, R3, D2 und C9 bis C12 bilden ein kleines Netzteil, das Gleichspannungen zwischen 7...30 Volt und Wechselspannungen zwischen 9...20 Volt sicher auf 5 Volt regelt.

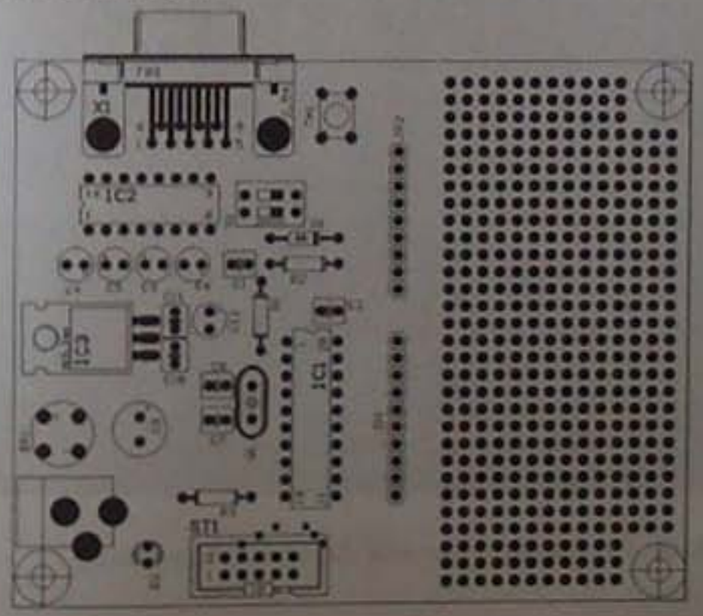


Bild 5.1b: Bestückungsplan der AVR-Experimentierplatine.

Stückliste

Widerstände:

R1, R3 = 220 Ω
R2 = 10 kΩ

Kondensatoren:

C1, C2, C10, C11 = 100 nF
C3...C6, C12 = 10 µF / 16 V
C7, C8 = 22 pF
C9 = 100 µF / 35 V

Halbleiter:

BR1 = Brückengleichrichter B40C1500
D1 = 1N4148
D2 = LED rot
IC1 = AT90S2313
IC2 = MAX232
IC3 = 7805

Außerdem:

Q1 = Quarz 4 MHz
TA1 = Taster
JP1 = Pfostenleiste 1 x 10polig
JP2 = Pfostenleiste 1 x 9polig
ST1 = Wannenstecker 2 x 5polig
S1 = DIL-Schalter 2 x Ein
X1 = Sub-D-Buchse 9polig
BU1 = Niederspannungsbuchse 2,1 mm Stift

Tabelle 5.1: *Stückliste zur AVR-Experimentierplatine.*

5.1 Touch-Memories

Touch-Memories von Dallas Semiconductor sind Speicher, die in einem knopfzellenähnlichen, runden Edelstahlgehäuse von 16 mm Durchmesser und einer Dicke von 3,1 mm oder 5,9 mm, je nach Typ, untergebracht sind. Ihre Typenbezeichnung lautet DS 19xx.

Die Datenübertragung von und zu den Touch-Memories erfolgt über ein 1-Draht-Protokoll. Die Energie für die Datenübertragung wird von der Datenleitung selber gewonnen (Parasitic Power). Ein auf dem Chip integrierter Multiplexer entwirrt die ankommenden Signale und sorgt dafür, daß der eigentliche Speicherchip die richtigen Signale erhält. Befindet sich im Touch-Memory nur ein ROM (DS 1990A), wird keine weitere Stromversorgung benötigt. Bei den Typen mit SRAM (DS 1991 bis 1996) ist noch eine 3 V-Lithiumzelle integriert. Ihre Lebensdauer wird vom Hersteller mit 10 Jahren angegeben. Die Chips sind in CMOS-Technologie ausgeführt.

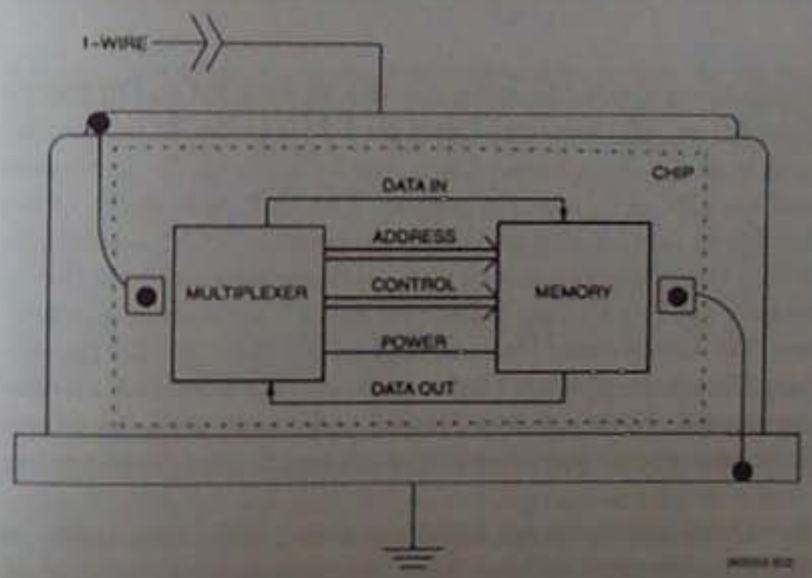


Bild 5.2: *Die Touch-Memories sind in einem runden Edelstahlgehäuse integriert. Ein Multiplexer entwirrt die Bits auf der 1-Draht-Datenleitung.*

Der Ausgangstreiber der Touch-Memories ist als Open-Drain ausgeführt. Dadurch ist es möglich, diese an jeder üblichen Hardware anzuschließen. Zudem kann der Open-Drain-Ausgang über einen Widerstand zu einem Wired-AND verknüpft werden, so daß sogar ein Zusammenschalten von mehreren Touch-Memories auf einem Bus möglich ist. Dallas nennt diesen Bus MicroLAN. Besitzt der verwendete Mikrocontroller keinen Open-Drain-Ausgang wird ein zusätzlicher Transistor notwendig und man muß zwei Ports für die Übertragung spendieren, so wie beim hier verwendeten AT90S2313 (siehe Schaltplan **Seite 208**).

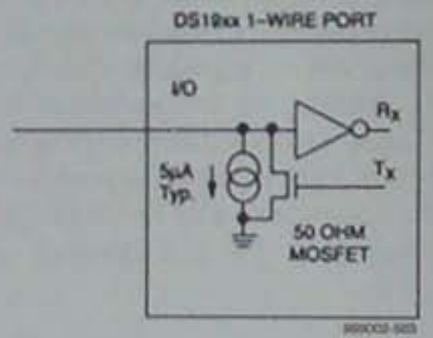


Bild 5.3: Der Open-Drain Ausgang ermöglicht ein Zusammenschalten von mehreren Touch-Memories auf dem Bus.

Dallas bietet Touch-Memories mit SRAM, EPROM und EEPROM Speicher an. Alle Versionen haben ein 8-Byte großes ROM gemeinsam, in dem sich ein Family Code, eine 6-Byte lange Seriennummer und ein CRC-8 (Cyclic Redundancy Check) Byte befindet. Dabei handelt es sich um kein maskenprogrammiertes ROM, sondern die 8-Byte lange Information wird nach der Herstellung durch das Durchtrennen von Polysilizium-Bahnen mit einem Laserstrahl eingeschrieben. Der Vorteil dieser Methode liegt auf der Hand: Bei der Herstellung können die gleichen Maskensätze, die sehr teuer sind, verwendet werden. Noch auf dem Siliziumwafer kann dann die unique Seriennummer für jedes Touch-Memory einzeln eingelasert werden.

Bei Touch-Memories mit SRAM ist eine 3 Volt-Lithiumzelle im Gehäuse integriert und man kann den Speicherinhalt beliebig oft verändern.

Bei den Typen mit EPROM (DS 198x Reihe) ist klar, daß man diese nur einmal programmieren kann, da das Gehäuse über kein Quarzglas-Fenster verfügt, um den Speicher wieder zu löschen. Dallas nennt diese Touch-Memories „Add-Only-Memory", d. h. man kann in ihnen nur zusätzliche Informationen in noch nicht programmierten Speicherzellen einbrennen. In diesem Abschnitt werden die Typen der DS 199x Reihe, die alle über SRAM verfügen, besprochen, da ihr Einsatz durch den immer wieder beschreibbaren Speicher interessanter erscheint.

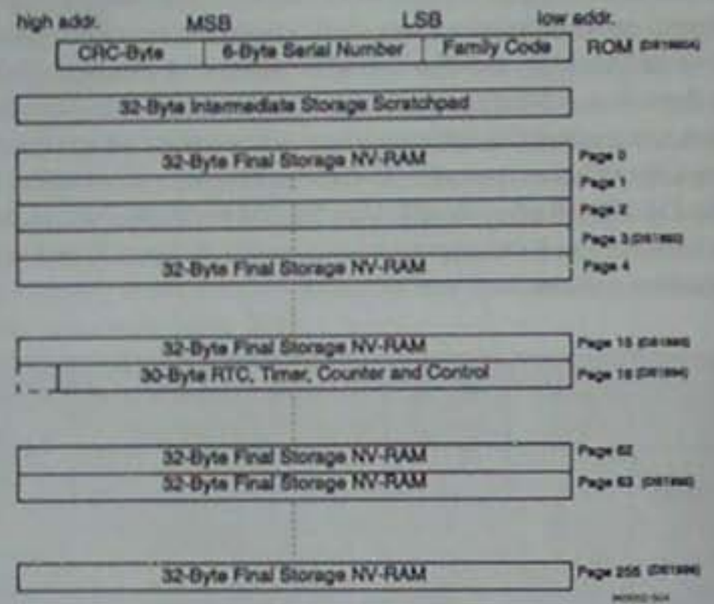
Tabelle 5.2: Übersicht der Touch-Memory Typen.

Typ	Family Code	Serien-Nr.	Speicher-größe	geschützter Speicher	RTC	Intervall-Zähler	Zyklus-Zähler
DS 1990A	01hex	ja	–	–	–	–	–
DS 1991	02hex	ja	512 Bit	3 x 384 Bit	–	–	–
DS 1992	08hex	ja	1 kBit	–	–	–	–
DS 1993	06hex	ja	4 kBit	–	–	–	–
DS 1994	04hex	ja	4 kBit	–	ja	ja	ja
DS 1995	0Ahex	ja	16 kBit	–	–	–	–
DS 1996	0Chex	ja	64 kBit	–	–	–	–

Befehle und Daten zu und von Touch-Memories werden Bit für Bit gesendet, bis der Befehl vollständig abgeschlossen ist. Das LSB-Bit wird dabei zuerst gesendet. Bei der Datenübertragung ist der Mikrocontroller der Master und das Touch-Memory der Slave. Die Synchronisation der Bits nimmt der Master durch eine High-Low-Flanke auf der Datenleitung vor. Danach tastet der Master oder der Slave die Datenleitung ab, je nach Befehl und Datenrichtung. Bei der Übertragung wird jedes Bit einzeln synchronisiert. So ist zwischen den Bits eine Pause erlaubt.

Das Lesen und Schreiben geschieht in sogenannten Time-Slots. Nach dem Berühren eines Touch-Memories sendet der Master üblicherweise einen Reset, um eine definierte Ausgangsposition zu schaffen. Dazu muß der Master die Datenleitung für mindestens 480 µs auf Low ziehen. Anschließend folgt eine ebenso lange Zeit, in der die Datenleitung auf High gehalten wird. Während dieser Zeit kann

Bild 5.4: Die Touch-Memories DS 1990A und DS 1992 bis DS 1996 verfügen über ein 8-Byte langes lasergeschriebenes ROM mit unique Seriennummer und unterschiedlich vielen Seiten SRAM.



das Touch-Memory einen Presence Pulse generieren. Dieser Pulse wird vom Touch-Memory nach der Zeit t_{RSTL} generiert und ist t_{RSTH} lang. Damit kann der Master feststellen, ob ein Touch-Memory am Bus angeschlossen ist oder, ob z. B. der Bus kurzgeschlossen ist.

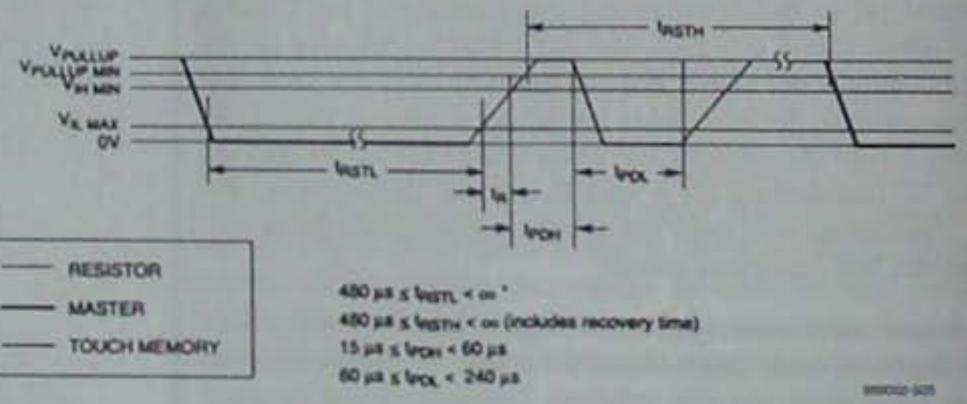


Bild 5.5: Für ein Reset muß der Master die Datenleitung für mindestens 480 µs auf Low ziehen. Daraufhin wird sich das Touch-Memory mit einem Presence-Pulse melden.

Die Übertragung von Bits zum Touch-Memory geschieht durch das Schreiben von 0- und 1-Bits im Write-Zero Time Slot und Write-One Time Slot, nachdem der Master mit einer High-Low Flanke die Übertragung initiiert hat. Der eigentliche aktive Teil dieser Slots ist 60 µs lang. Das Touch-Memory tastet die Datenleitung in der Mitte dieser Time-Slots ab, gerechnet von der abfallenden Flanke der Synchronisation.

Da der Hersteller für das Abtasten der Datenleitung durch das Touch-Memory einen Toleranzbereich zwischen 15 und 60 µs angibt, muß während dieser Zeit die Datenleitung stabil gehalten werden. Möchte man eine „1" übertragen, muß spätestens 15 µs nach der Synchronisation die Datenleitung auf High gezogen werden.

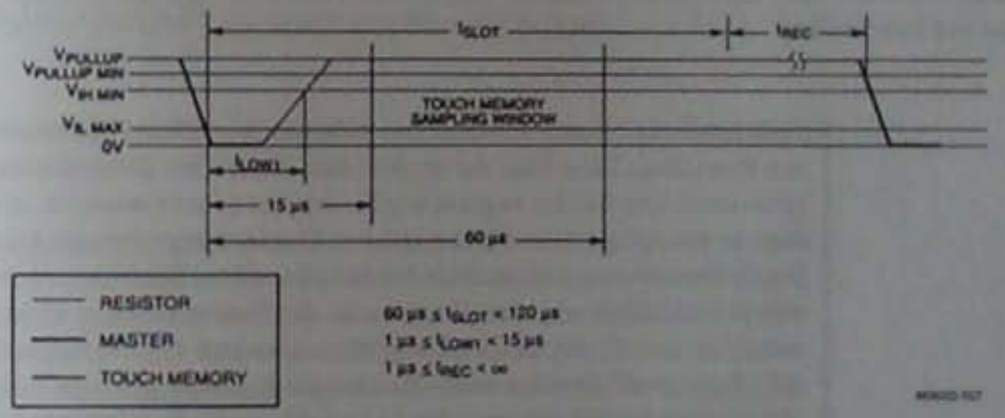


Bild 5.6: Eine „1" schreibt der Master, indem er die Datenleitung nach der High-Low-Flanke nach maximal 15 µs wieder losläßt.

Zum Senden einer „0" muß die Datenleitung mindestens 60 µs auf Low gehalten werden. Nach der Bit-Übertragung benötigt das Touch-Memory eine Erholzeit von mindestens 1 µs, bevor die Übertragung fortgesetzt werden kann.

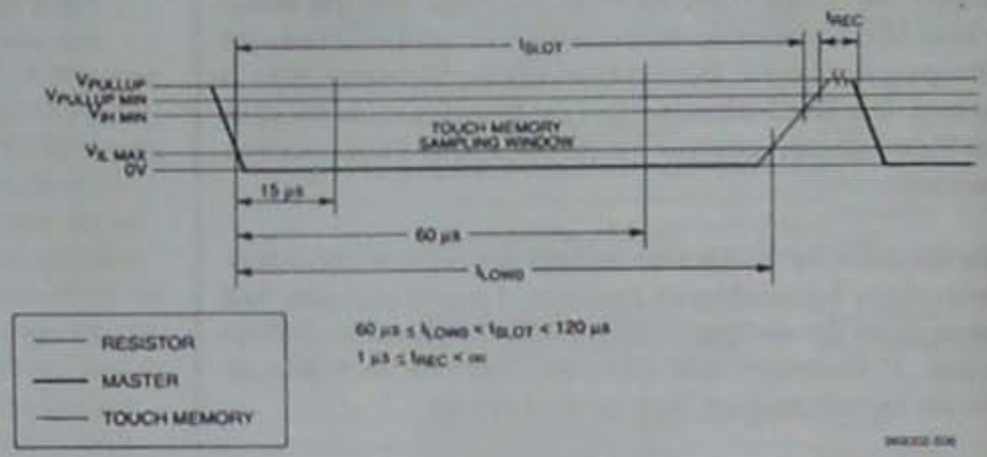


Bild 5.7: Um eine „0" zu schreiben, muß der Master die Datenleitung für mindestens 15 µs auf Low halten.

Zum Lesen von Daten aus dem Touch-Memory muß der Master einen Read-Data Time Slot, der ähnlich dem Write-One Time Slot ist, generieren. Der Master beginnt wieder mit der Synchronisation, indem er eine High-Low-Flanke auf der Datenleitung erzeugt. Das Touch-Memory sendet darauf ein Bit des adressierten Speichers. Handelt es sich dabei um eine „1", braucht das Touch-Memory nichts weiter zu tun, da die Datenleitung über einen Pull-Up-Widerstand auf High-Level gehalten wird. Handelt es sich hingegen um eine „0", zieht es die Datenleitung für 15 µs auf Low. In dieser Zeit kann der Master die Leitung abtasten. Der Master muß nach der Synchronisationsflanke die Leitung für mindestens 1 µs auf Low halten. Sie sollte aber nicht länger als nötig Low sein, um das Abtastfenster möglichst lang zu halten. Nach der Übertragung benötigt das Touch-Memory zwischen 0 und 45 µs, um die Datenleitung wieder loszulassen.

Befehle und Daten zum Touch-Memory hin werden durch die Aneinanderreihung von Write-Zero und Write-One Time Slots übertragen. Zum Lesen von Daten und Rückmeldungen des Touch-

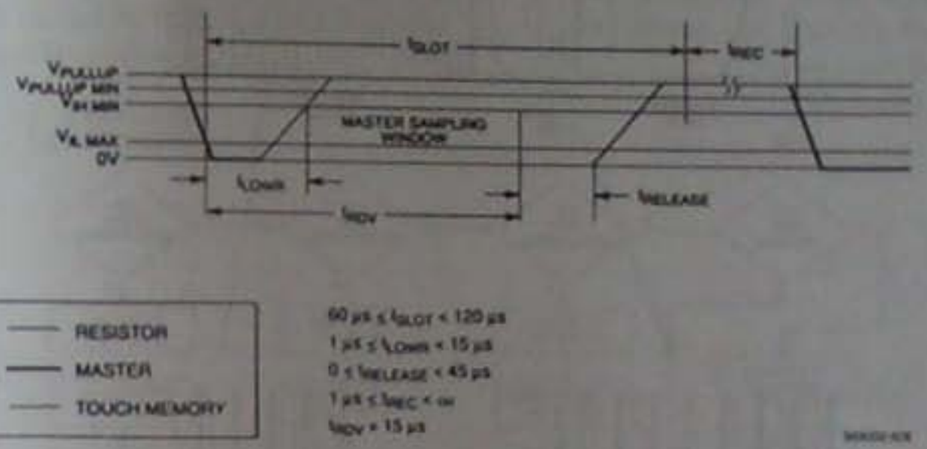


Bild 5.8: Der Master tastet innerhalb von 15 µs nach der High-Low-Flanke die Datenleitung ab, um Daten vom Touch-Memory zu lesen.

Memories muß der Master entsprechend viele Read-Data Time Slots erzeugen.

Insgesamt ist das Lesen und Schreiben von Touch-Memories sehr trickreich gestaltet. Es wird hier kurz der Ablauf zum Schreiben von Daten ins Touch-Memory skizziert: Nach dem Write Scratchpad-Befehl muß der Master eine 2-Byte lange Target-Adresse TA1 und TA2 übertragen, die die Speicherstelle im Touch-Memory angibt, ab welcher z. B. das Schreiben stattfinden soll. Dazu haben Touch-Memories sogenannte Adreß-Register. Die ersten fünf Bit dieser Target-Adresse geben den Offset innerhalb einer 32-Byte Seite an. Die zu schreibenden Daten folgen dieser Adresse. Der Master beendet das Schreiben mit einem Reset. Jetzt befinden sich die Daten im Scratchpad. Um die übertragenen Daten zu verifizieren, den End-Offset der Daten und den Status des Touch-Memories zu erhalten, müssen die Daten mit dem Read Scratchpad-Befehl wieder aus dem Touch-Memory gelesen werden. Den End-Offset und den Status speichert das Touch-Memory im E/S-Register, das nur einen Read-Only Zugriff erlaubt. Die ersten fünf Bit geben die Endadresse der

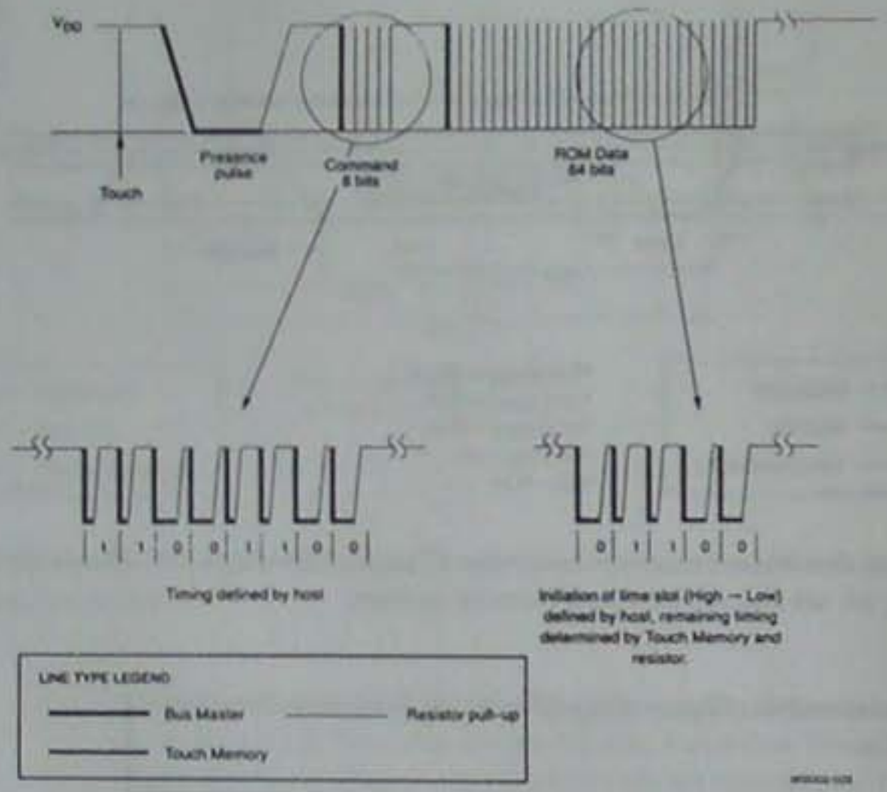
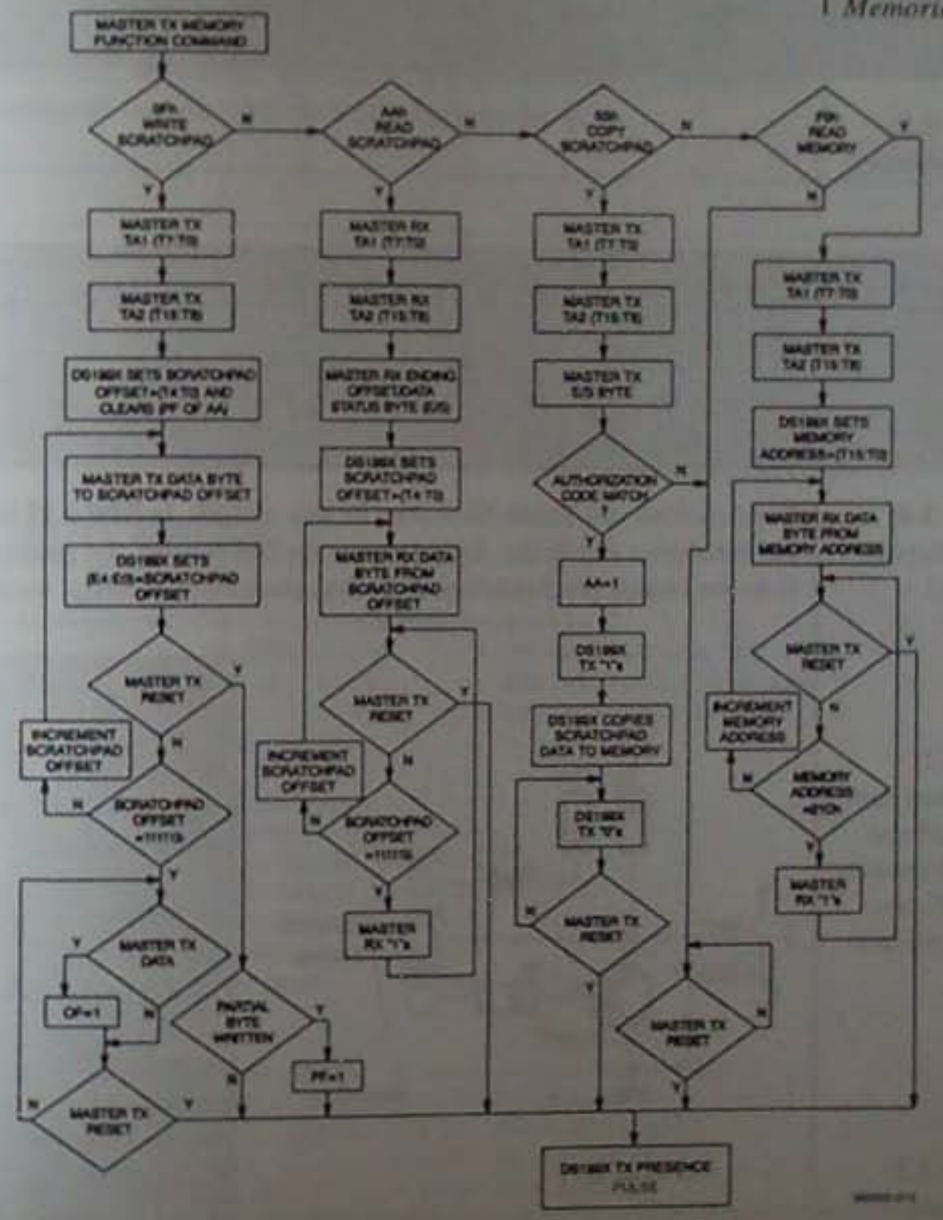


Bild 5.9: *Nachdem sich das Touch-Memory mit einem Presence Pulse gemeldet hat, überträgt der Master den Befehl das ROM zu lesen. Das Touch-Memory antwortet mit seinem ROM-Inhalt.*

geschriebenen Daten an. Hat man versucht über die 32. Stelle der Seite zu schreiben, wird das Overflow Flag (OF) gesetzt. Ist ein Byte unvollständig übertragen worden, so setzt das Touch-Memory das Partial Byte Flag (PF). Setzt das Touch-Memory eines dieser Flags, so ist der Schreibversuch mißlungen. Das siebte Bit in diesem Register ist das Authorization Accepted Flag (AA). Dieses ist wichtig zu behalten, da man es zum Kopieren des Scratchpad-Inhalts zur endgültigen Stelle im Touch-Memory benötigt. Nur wenn weder OF- noch PF-Flag gesetzt sind, wird dieses Flag gesetzt. Hat man diese Bits gesammelt und sich gemerkt, kann man den Befehl „Copy Scratchpad" geben, gefolgt von TA1, TA2 und E/S Inhalt. Dieser Inhalt stellt den Authorization Code dar. Nur wenn dieser

übereinstimmt, setzt das Touch-Memory das AA-Flag und der Inhalt des Scratchpads wird an die gewünschte Stelle kopiert. In **Bild 5.10** ist das vollständige Flußdiagramm für die Ansteuerung von Touch-Memories angegeben.

Bild 5.10: *Flußdiagramm für die Funktionen der Touch-Memories.*



ROM				Scratchpad		Memory	
Read	Skip	Match	Search	Read	Write	Copy	Read
33h	CCh	55h	F0h	AAh	0Fh	55h	F0h

Tabelle 5.3: Touch-Memory Befehle.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Bemerkung
T7	T6	T5	T4	T3	T2	T1	T0	Target Address TA1
T15	T14	T13	T12	T11	T10	T9	T8	Target Address TA2
AA	OF	PF	E4	E3	E2	E1	E0	Ending Address/Data Status (E/S)

Tabelle 5.4: Adreß-Register TA1, TA2 und E/S.

Der Anschluß von Touch-Memories ist sehr einfach. In **Bild 5.11** ist der Schaltplan abgebildet. Dabei ist nur der Teil gezeigt, der zusätzliche zur bereits beschriebenen Experimentierplatine benötigt wird.

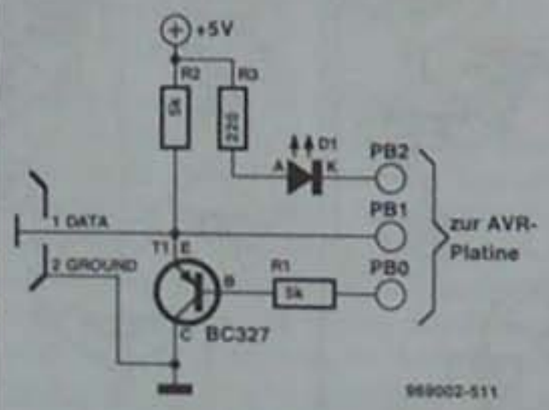


Bild 5.11: Schaltplan zum Anschluß von Touch-Memories an die Experimentierplatine.

Stückliste:

Widerstände:
R1, R2 = 5 kΩ
R3 = 220 Ohm

Halbleiter:
T1 = BC327
D1 = Leuchtdiode

Außerdem:
Touch-Memory Halter DS9094F von Dallas

Tabelle 5.5: Stückliste.

Das Beispielprogramm schreibt das Byte „A5" in die Adresse 0001 des Touch-Memorys. Anschließend wird die gleiche Adresse wieder ausgelesen. Damit sind die beiden wichtigsten Funktionen, Lesen und Schreiben, demonstriert. Bevor das Programm versucht zu Schreiben oder zu Lesen, prüft es, ob ein Touch-Memory angeschlossenen ist. Ist ein Touch-Memory angeschlossen, wird die Funktion ausgeführt. Ansonsten signalisiert die LED, daß kein Touch-Memory verfügbar ist, oder, daß ein Fehler aufgetreten ist.

```
;***************************************************************
; Touch-Memory Ansteuerung
;
; Version 1.0
;
; File-Name: TM.ASM
; AT90S2313 mit 4 MHz Takt
;
;***************************************************************

.device AT90S2313
.include „2313def.inc"

;Konstanten fuer Touch-Memory Interface
.equ        TM_OUT  = PB0       ; PB0 sendet zum TM
.equ        TM_IN   = PB1       ; PB1 empfaengt vom TM
.equ        LED     = PB2       ; PB2 LED fuer Fehler


;Touch-Memory Befehle
.equ        RD_ROM          = 0x33
.equ        WR_SCRATCH      = 0x0F
.equ        RD_SCRATCH      = 0xAA
.equ        COPY_SCRATCH    = 0x55
.equ        RD_MEMORY       = 0xF0
.equ        SKIP_ROM        = 0xCC

;Variablen
.def        delay_cntr      = r16; Schleifenzaehler
.def        buffer          = r17; Datenbyte
.def        data_cntr       = r18; Datenbitzeiger
.def        hilf            = r19; Hilfsregister
.def        off_es          = r20; End Adresse
```

```
.def        high_adr    = r21; 2. Adress-Byte
.def        low_adr     = r22; 1. Adress-Byte
.def        datal       = r23; Datenbyte
.def        temp        = r24; temporaeres Register


.CSEG
.ORG        0x00                ; Programm beginnt bei 0
            rjmp    main        ; Starte Hauptprogramm


;***************************************************************
; Subroutine init
; Initialisiere Stack-Pointer, Port und Timer0. Loesche LED.
;***************************************************************

init:       sbi     PORTB,TM_OUT
            ldi     temp,0b11111101 ; PB0 und PB2 Ausg., PB1 Eing.
            out     DDRB,temp
            ldi     temp,0b00000010 ; Timer0-Vorteiler CK/8
            out     TCCR0,temp
            sbi     PORTB,LED       ; Loesche LED
            ret

;***************************************************************
; Subroutine init_tm
; Initialisiere Touch-Memory und teste auf „Presence Pulse"
; Uebergibt T-Flag=1, falls TM da
;           T-Flag=0, falls TM nicht da
;***************************************************************

init_tm:    clr     delay_cntr      ; Timer0 initialisieren
            out     TCNT0,delay_cntr

            cbi     PORTB,TM_OUT    ; Bus auf low
reset_delay: in     temp,TCNT0      ; halte Bus min. 480us
            cpi     temp,245        ; auf low
            brlo    reset_delay

            sbi     PORTB,TM_OUT    ; Datenleitung loslassen

            ldi     delay_cntr,20   ; warte min. 15 us
wait_delay: dec     delay_cntr      ; damit TM Bus anzieht
            brne    wait_delay
```



```
            set                     ; nehme an, TM ist da

            ldi     hilf,5          ; teste 5mal, ob TM da
test_tm:    sbis    PINB,TM_IN      ; falls TM da, springe raus
            rjmp    tm_ok           ;
            ldi     delay_cntr,20   ; warte min. 15 us
wait_delay2: dec    delay_cntr      ; und teste nochmal
            brne    wait_delay2
            dec     hilf
            brne    test_tm         ; teste 5 mal, ob TM da

            clt                     ; TM doch nicht da! Loesche CARRY

tm_ok:      clr     delay_cntr      ; Schleife fuer 480 us
            out     TCNT0,delay_cntr; zur Erholung des TMs
release:    in      temp,TCNT0      ;
            cpi     temp,245        ;
            brlo    release

            sbis    PINB,TM_IN      ; teste nach 480us Bus low?
            clt                     ; Bus permanent auf low, Fehler!
            ret


;***************************************************************
; Subroutine cmd_out
; Subroutine cmd_out sendet Befehl bestehend aus 8 Bit zum
; Touch-Memory. Es wird ueber Pin PB0 zum Touch-Memory
; gesendet und ueber Pin PB1 vom Touch-Memory gelesen.
; Command muss im Register „buffer" uebergeben werden.
; Benutzt „data_cntr", „buffer", „delay_cntr"
;***************************************************************

cmd_out:    ldi     data_cntr,8     ; Command ist 8 Bit
cmd_bit:    cbi     PORTB,TM_OUT    ; Startzeichen
            ldi     delay_cntr,3    ; halte Bus 2 us auf low
cmd_delay1: dec     delay_cntr
            brne    cmd_delay1
            ror     buffer          ; schiebe Bit ins CARRY
            brcc    wr_zero         ; Ist Bit „0"?
            sbi     PORTB,TM_OUT    ; Nein, Bit war eine „1"
wr_zero:    ldi     delay_cntr,80   ; halte Bus fuer 60 us
cmd_delay:  dec     delay_cntr
            brne    cmd_delay
```

```
            sbi     PORTB,TM_OUT    ; lasse nach 60 us los
            nop
            nop
            dec     data_cntr       ; wiederhole alle 8 Bit
            brne    cmd_bit
            nop
            nop
            ret


;***************************************************************
; Subroutine data_in
; Subroutine data_in liest 8 Bit vom Touch-Memory ueber Pin PB1
; ein.
; Ueber Pin PB0 wird zum Touch-Memory gesendet. Das eingelesene
; Byte steht in „buffer".
; Benutzt „buffer", „delay_cntr", „data_cntr"
;***************************************************************

data_in:    ldi     data_cntr,8     ; 8 Bit aus TM lesen
read_bit:   cbi     PORTB,TM_OUT    ; Startzeichen
            nop                     ; warte 1 us
            nop
            nop
            nop
            sbi     PORTB,TM_OUT    ; lasse Bus los
            ldi     delay_cntr,8    ; warte ca. 6 us
in_delay:   dec     delay_cntr
            brne    in_delay
            sec                     ; nehme an, habe „1" gelesen
            sbis    PINB,TM_IN
            clc                     ; nein, war eine „0"
            ror     buffer          ; schiebe Bit in „buffer"
            ldi     delay_cntr,30   ; warte ca. 15 us
rd_rel_delay:dec    delay_cntr
            brne    rd_rel_delay
            dec     data_cntr
            brne    read_bit
            ret

;***************************************************************
; Subroutine fehler
; Sollte ein Fehler auftreten oder kein Touch-Memory
; angeschlossen sein, so wird das ueber die LED angezeigt.
;***************************************************************
```



```
fehler:     cbi     PORTB,LED       ; LED einschalten
            rjmp    fehler          ; Endlosschleife



;***************************************************************
; Hauptprogramm
; Schreibt 1 Byte ins Touch-Memory uns liest es wieder aus.
; Start-Adresse muss in „low_adr" und „high_adr" uebergeben
; werden. Das Byte, das geschrieben werden soll, muss im
; W-register uebergeben werden.
; Das gelesene Byte steht in „buffer".
; Sollte ein fehler auftreten oder kein Touch-Memory
; angeschlossenen sein, wird die LED eingeschaltet.
;***************************************************************

main:       ldi     temp,RAMEND     ; setze Stack-Pointer
            out     SPL,temp        ; an das SRAM-Ende

            rcall   init

            ldi     low_adr,0x01    ;Byte A5h in Adresse
                                    ; 0001h schreiben
            ldi     high_adr,0x00
            ldi     data1,0xA5

            rcall   init_tm         ; ist TM da?
            brtc    fehler          ; ja, springe weiter
                                    ; nein -> Fehler!

; Hier beginnt das Schreiben des Touch/Memory
            ldi     buffer,SKIP_ROM     ; ueberspringe ROM
            rcall   cmd_out             ; sende skip_rom command
            ldi     buffer,WR_SCRATCH   ; bereite wr_scratch vor
            rcall   cmd_out             ; sende wr_scratch command
            mov     buffer,low_adr      ; Startadresse (low) senden
            rcall   cmd_out             ; Adr zum TM uebertragen
            mov     buffer,high_adr     ; Startadresse (high) senden
            rcall   cmd_out             ; Adr zum TM uebertragen
            mov     buffer,data1        ; Byte zum TM uebertragen
            rcall   cmd_out
            rcall   init_tm             ; Uebertragung beenden
```

```
        ; Hier beginnt der Datenvergleich
        ldi     buffer,SKIP_ROM         ; ueberspringe ROM
        rcall   cmd_out
        ldi     buffer,RD_SCRATCH       ; lies Scratchpad
        rcall   cmd_out
        rcall   data_in                 ; low Adresse lesen
        cp      buffer,low_adr          ; low Adresse vergleichen
        brne    fehler                  ; Vergleich negativ, Fehler!
        rcall   data_in                 ; high Adresse lesen
        cp      buffer,high_adr         ; high Adresse vergleichen
        brne    fehler                  ; Vergleich negativ, Fehler!
        rcall   data_in                 ; lies OFFSET/ES
        mov     off_es,buffer           ; OFFSET/ES sichern

        ; Nach Datenvergleich werden die Daten aus dem
        ; Scratchpad in die endgueltige Speicherstelle
        ; kopiert
        rcall   init_tm
        ldi     buffer,SKIP_ROM
        rcall   cmd_out
        ldi     buffer,COPY_SCRATCH
        rcall   cmd_out
        mov     buffer,low_adr
        rcall   cmd_out
        mov     buffer,high_adr
        rcall   cmd_out
        mov     buffer,off_es
        rcall   cmd_out
        rcall   init_tm

        ; Lesen
        rcall   init_tm
        ldi     buffer,SKIP_ROM
        rcall   cmd_out
        ldi     buffer,RD_MEMORY
        rcall   cmd_out
        mov     buffer,low_adr
        rcall   cmd_out
        mov     buffer,high_adr
        rcall   cmd_out
        rcall   data_in                 ; Inhalt steht in buffer
        out     PORTB,buffer

loop:   rjmp    loop                    ; Endlosschleife
```


5.2 Ansteuerung von EEPROMs mit I²C-Bus-Protokoll

Der I²C-Bus ist ein serieller Bus, der die Kommunikation zwischen Integrierten-Schaltungen ermöglicht. Entwickelt wurde dieser Bus von der Firma Philips, die auch das Patent dafür hält. Andere Firmen, die diesen Bus in ihren Bauelementen implementieren, müssen von der Firma Philips eine Lizenz erwerben. Anwendungen findet dieser Bus u. a. bei seriellen EEPROMs (Electrically Erasable and Programmable Read Only Memory).

Die Kommunikation auf diesem Bus erfolgt über die Leitungen SDA (Serial Data) und SCL (Serial Clock). Alle Devices, also Mikrocontroller, Speicher und Peripherie-Bausteine, sind über diese beiden Leitungen parallel geschaltet. Die Datenleitung SDA muß stabil sein während die Taktleitung SCL auf logisch High ist. In diesem Zeitintervall liegen gültige Daten auf der Datenleitung. Der Pegel von SDA darf sich nur ändern, während SCL auf logisch Low liegt. Jedes Device hat eine eindeutige Adresse (Device Select Code), um auf dem Bus angesprochen werden zu können. Ein Device kann Sender (Transmitter) und Empfänger (Receiver) sein, also Daten zu einem anderen Device senden oder von diesem empfangen. Ein Device, das eine Datenübertragung initiiert und den Takt auf der SCL-Leitung generiert, wird als Master bezeichnet (i. a. wird das ein Mikrocontroller sein). Zu diesem Zeitpunkt sind alle anderen Devices Slaves. In **Tabelle 5.6** sind diese Begriffe zusammengefaßt.

Begriff	Erläuterung
Master	Ein Device, welches die Datenübertragung initiiert, den Takt generiert und die Übertragung wieder beendet.
Receiver	Ein Device, welches Daten vom Bus empfängt.
Slave	Ein Device, welches von einem Master angesprochen wird.
Transmitter	Ein Device, welches Daten auf dem Bus sendet.

Tabelle 5.6: Begriffsdefiniton beim I²C-Bus.

Da sich an einem I²C-Bus mehrere Master befinden können, wird dieser auch als Multi-Master-Bus bezeichnet. Damit es auf dem Bus zu keinen Datenkonflikten kommt, da gleichzeitig mehrere Master eine Datenübertragung initiieren könnten und damit die Ausgangsstufen der Devices beschädigt werden könnten, handelt es sich beim I²C um einen Wired-AND-Bus, d. h. alle Ausgangsstufen sind als Open-Drain bzw. Open-Collector, je nach verwendete Halbleitertechnologie, ausgeführt. Die Leitungen SDA und SCL sind über Pull-Up-Widerstände an die positive Versorgungsspannung V_{CC} gelegt. In **Bild 5.12** ist die Verbindung von zwei Devices auf dem Bus dargestellt.

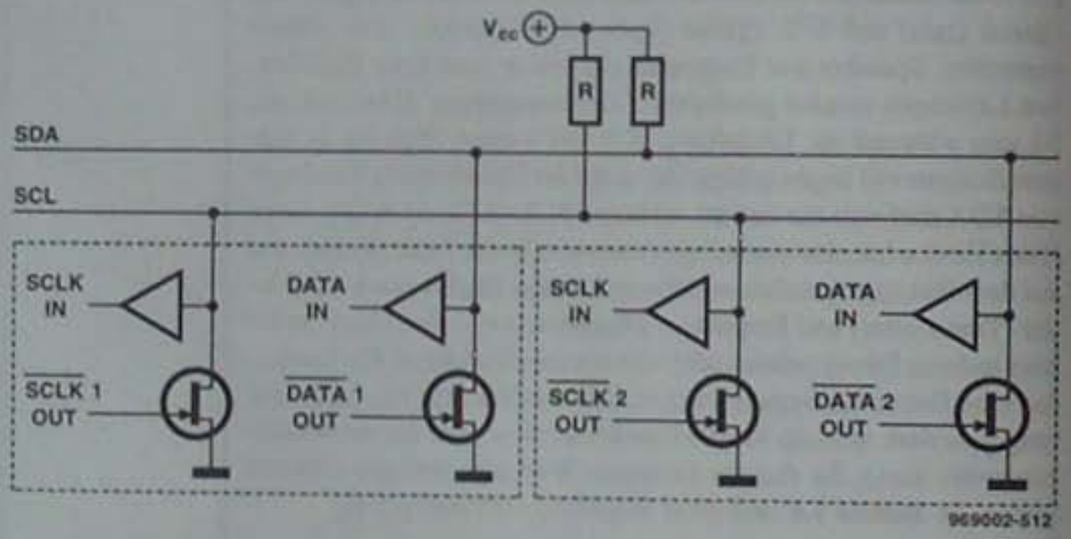


Bild 5.12: Verbindung von zwei Devices mit Open-Drain Ausgangsstufen auf dem I²C-Bus. Die Leitungen SDA und SCL werden über Pull-Up-Widerstände auf die positive Versorgungsspannung gehalten.

Der Master initiiert die Datenübertragung mit einer Startsequenz und beendet diese mit einer Stopsequenz. Dabei ist eine High-Low-Flanke auf der SDA-Leitung, während SCL auf High liegt, die Start- und eine Low-High Flanke die Stopsequenz (siehe **Bild 5.13**). Nun wird auch klar, weshalb die Datenleitung während SCL High ist, stabil

sein muß, da ansonsten fälschlicherweise eine Start- oder Stopsequenz erkannt würde.

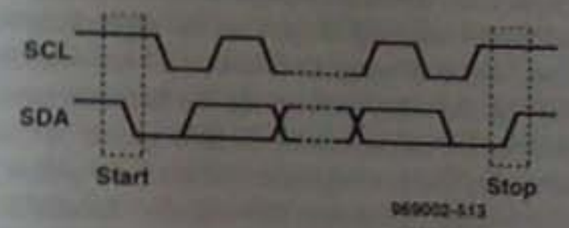


Bild 5.13: Start- und Stopsequenz beim I²C-Bus

Nachdem der Master die Datenübertragung mit einer Startsequenz begonnen hat, sendet der Transmitter die Daten. Jedes Datenbyte muß 8 Bit lang sein. Die serielle Übertragung des Datenbytes erfolgt mit dem MSB (Most Significant Bit) zuerst. Nachdem alle 8 Bits übertragen worden sind, muß der Receiver den Datenempfang mit einem Acknowledge (ACK)-Bit quittieren. Das geschieht dadurch, indem der Transmitter die SDA-Leitung losläßt und der Receiver diese auf logisch Low zieht. Der Master muß für das ACK-Bit ein neuntes Taktsignal generieren. Während dieser Zeit tastet der Transmitter die SDA-Leitung nach dem ACK-Bit ab. Ist die Datenübertragung beendet, generiert der Master eine Stopsequenz. Die Zeitverläufe dieser Kommunikation sind in **Bild 5.14** verdeutlicht.

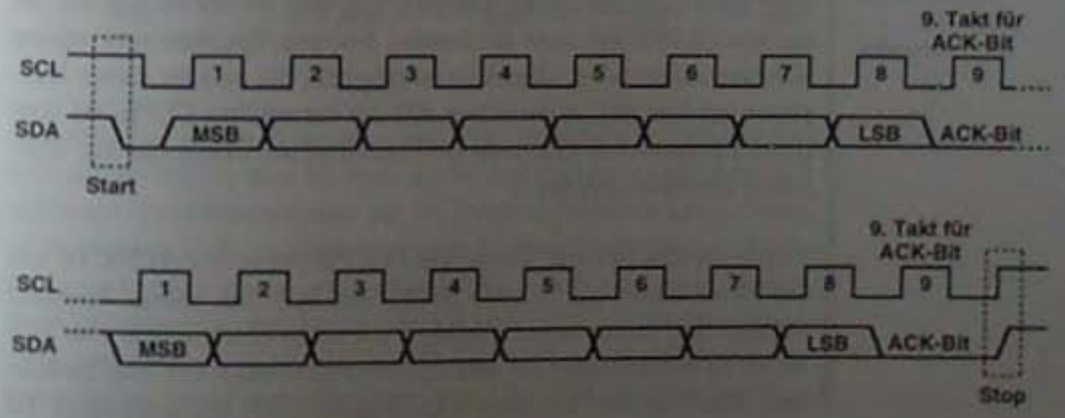


Bild 5.14: Datenübertragung auf dem I²C-Bus.

Gezeigt ist die Startsequenz gefolgt vom ersten Datenbyte und dazugehörigem ACK-Bit, sowie das letzte Datenbyte mit ACK-Bit und abschließender Stopsequenz.

Das erste Byte nach der Startsequenz ist die Adresse des Slaves. Diese Adreßbyte ist wie folgt aufgebaut: Die eigentliche Adresse ist 7 Bit lang (a0...a6) und belegt die Bits b1...b7 des Adreßbytes. Das Bit b0 (LSB) des Adreßbytes wird für die Richtungsanzeige R/W (Read/Write) der Daten verwendet. Ist R/W gleich „1", so liest der Master Daten vom Slave. Hingegen wird mit R/W gleich „0" angezeigt, daß der Master Daten zum Slave sendet. **Tabelle 5.7** faßt die Bedeutung der einzelnen Bits des Adreßbytes nocheinmal zusammen.

Tabelle 5.7: Aufbau des Adreßbytes beim I²C-Bus.

MSB							LSB
b7	b6	b5	b4	b3	b2	b1	b0
a6	a5	a4	a3	a2	a1	a0	R/W*

*R/W = „1": Master liest vom Slave
R/W = „0": Master schreibt zum Slave

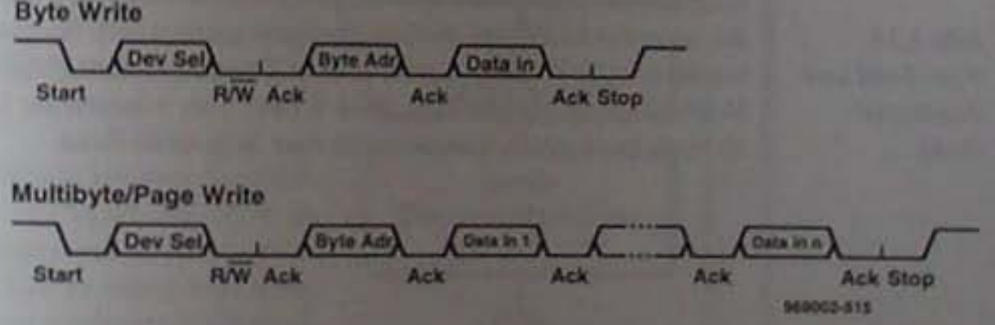
Die Bits b4 bis b7 des Adreßbytes bei EEPROMs haben die Kodierung 1010 (Device Code), also Ah. Die Bits b1 bis b3 werden bei einigen EEPROMs dazu verwendet, um dem Baustein eine eigene Adresse zuzuordnen. Damit ist es möglich, bis zu acht dieser Bausteine auf dem Bus zu betreiben. Ob das verwendete EEPROM über diese Fähigkeit verfügt, muß im Einzelfall im Datenblatt des Bausteins überprüft werden.

Das in diesem Beispiel verwendete EEPROM ist ein AT24C16 von der Firma Atmel mit 16 kbit (2048 Byte) Speicher. Dieser Baustein verfügt über keine Adreßeingänge wie oben beschrieben. Die Bits b1 bis b3 werden ignoriert und brauchen nicht berücksichtigt zu werden. Pin 7 ist der Schreibschutz-Eingang (WP, Write-Protect). Ist

WP an Masse gelegt, so läßt sich der Speicher ganz normal lesen und beschreiben. Wird WP mit V_{CC} verbunden, läßt der Speicher sich zwar noch lesen, aber die obere Hälfte nicht mehr beschreiben.

Möchte man eine bestimmte Speicherstelle im EEPROM programmieren geht man wie folgt vor (vgl. **Bild 5.15a**): Der Master (Mikrocontroller) initiiert die Datenübertragung mit einer Startbedingung. Anschließend muß dieser noch das Device-Select-Byte (Adreßbyte) senden. Dieses Byte besteht aus der Folge 1010 (Device Code), drei Block-Bits, die einen der acht Blöcke von 256 Byte adressieren, und schließlich das R/W-Bit, das zum Schreiben „0" sein muß. Das EEPROM quittiert das empfangene Byte mit einem ACK-Bit. Daraufhin sendet der Master die Byte-Adresse (die unteren acht Bits), die wiederum durch das EEPROM quittiert werden. Nun ist die Speicherstelle im EEPROM adressiert und der Master kann das eigentlich Datenbyte senden. Nachdem der Master das ACK-Bit des EEPROMs empfangen hat, startet er die Programmierung, indem er eine Stopbedingung generiert.

Bild 5.15: Byte-Write und Multibyte/Page-Write.

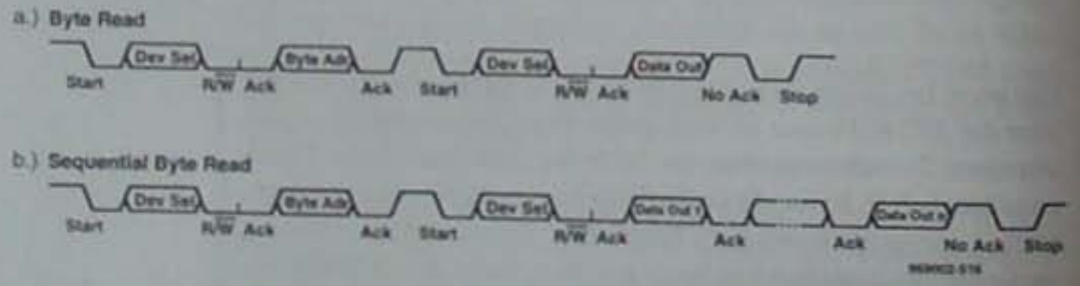


Um nicht bei jedem Byte diese Prozedur ausführen zu müssen, besitzt der AT24C16 eine 16 Byte große Page. Dabei handelt es sich um einen Zwischenspeicher, der 16 Bytes aufnehmen kann. Damit ist es möglich 16 Bytes, die im Speicher hintereinander liegen, gleichzeitig zu programmieren (vgl. **Bild 5.15b**). Dazu sendet der Master, im Gegensatz zum oben beschriebenen Byte-Write, keine Stopbedingung nach dem ACK-Bit des Datenbytes, sondern sendet bis zu sieben weitere Datenbytes, die immer vom EEPROM quittiert werden.

Erst nachdem der Master das ACK-Bit des letzten Datenbytes empfangen hat, sendet er eine Stopbedingung, um den Programmiervorgang zu starten. Diese Methode nennt man Multibyte-Write, falls man mehrere Bytes überträgt und Page-Write, falls man so viele Bytes überträgt, wie die Page lang ist.

Möchte man ein bestimmtes Byte aus dem EEPROM lesen, handelt es sich um ein Byte-Read (vgl. **Bild 5.16a**). Diese Prozedur gleicht bis zur Byte-Adresse und dem anschließenden ACK-Bit einem Byte-Write. Tatsächlich ist es auch ein Byte-Write, da die Byte-Adresse ins Register des EEPROMs geschrieben werden muß. Abweichend zum Byte-Write muß nun beim Read eine zweite Startbedingung gesendet werden, um ein weiteres Device Select-Byte einzuleiten. Bei diesem zweiten Device-Select-Byte muß das R/W-Bit „1" sein. Nachdem das EEPROM dieses Byte mit einem ACK-Bit quittiert hat, sendet es das Datenbyte. Das empfangene Datenbyte wird vom Master nicht quittiert, stattdessen wird für die Zeitdauer des ACK-Bits die SDA-Leitung auf High gehalten. Durch eine anschließende Stopbedingung beendet der Master die Datenübertragung. Sendet hingegen der Master nach dem Empfang eines Datenbytes ein ACK-Bit, setzt das EEPROM die Datenausgabe solange fort, bis der Master kein ACK-Bit mehr sendet und die Datenübertragung mit einer Stopbedingung beendet (vgl. **Bild 5.16b**). Das wiederholte Lesen ab einer bestimmten Adresse nennt man Sequential-Read.

Bild 5.16: Byte-Read und Sequential-Read.



Das Programmieren einer EEPROM-Zelle dauert typisch zwischen 2 ms und 10 ms, je nach Bauteiltyp, da verschiedene Hersteller unterschiedliche Halbleitertechnologien anwenden. Bevor nun an einem EEPROM seine Programmierung nicht vollständig abgeschlossen ist, reagiert es nicht auf eine erneute Datenübertragung. Der Mikrocontroller müßte also die maximal mögliche Programmierzeit, die der Hersteller angibt abwarten, bevor er eine neue Datenübertragung initiiert. Um diesen Leerlauf des Mikrocontrollers zu vermeiden, ist bei EEPROMs ein Acknowledge-Polling (eine Art Anfrage) implementiert. Statt die Programmierzeit abzuwarten, die zudem noch von äußeren Einflüsse wie Temperatur usw. abhängt, generiert der Mikrocontroller eine neue Startbedingung und sendet ein Device-Select-Byte zum Schreiben (R/W=0). Sollte das EEPROM mit der Programmierung fertig sein, sendet es ein ACK-Bit und der Mikrocontroller setzt die Übertragung fort. Andernfalls ignoriert das EEPROM die Anfrage und der Mikrocontroller startet eine neue Anfrage (vgl. **Bild 5.17**).

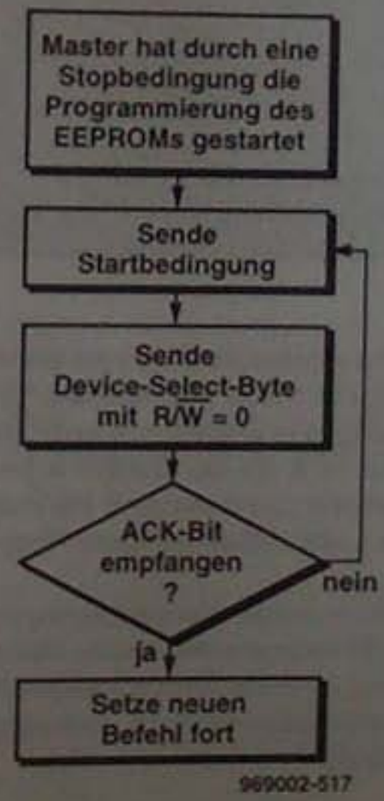


Bild 5.17: Acknowledge-Polling.

Der Schaltplan zum Anschluß eines EEPROMs AT24C16 an die Experimentierplatine ist in **Bild 5.18** zu sehen. Hier übernehmen wieder ein Widerstand (R3) und ein Transistor (T1) die Aufgabe eines Open-Collektor Ausgangs, um kompatibel zum Ausgang des EEPROMs zu sein.

Bild 5.18: Anschluß eines seriellen EEPROMs an die Experimentierplatine.

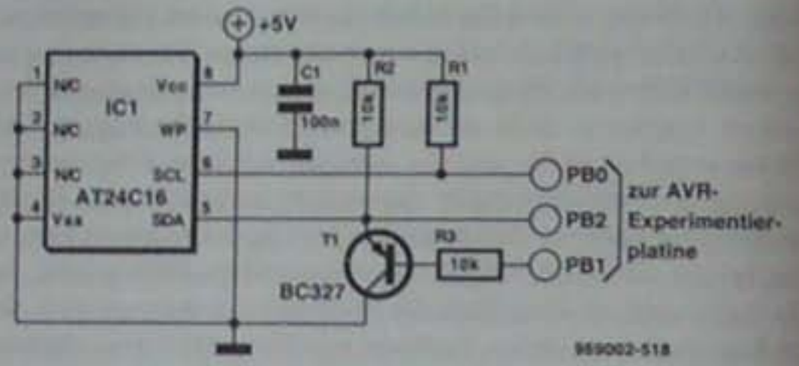


Tabelle 5.8: Stückliste.

Stückliste:

Widerstände:
R1...R3 = 10 kOhm

Halbleiter:
IC1 = EEPROM AT24C16
T1 = PNP-Transistor BC327 o.ä.

Kondensatoren:
C1 = 100 nF

Das abgedruckte Programm schreibt ein Byte in das EEPROM und liest es wieder aus. Nachdem für die Programmierung die Stopbedingung gesendet worden ist, wird mit Acknowledge-Polling auf das ACK-Bit des EEPROMs gewartet, bevor die Programmausführung fortgesetzt wird. Die einzelnen Teile des Programms können sehr einfach für eigene Programme herangezogen werden.

Die maximale Datenübertragungsrate beim I²C-Bus beträgt zwischen 100 kbit/s und 400 kbits/s. Dadurch dürfte es keine Probleme geben, die Schnittstelle softwaremäßig zu implementieren. Da der Bus vollständig statisch ist, wäre sogar eine Taktfrequenz von 1 Hz zulässig.


```
;***************************************************************
; Programm 'i2cbus.src'
; Das Programm i2cbus.src steuert serielle EEPROMs mit I2C-Bus
; an.
; Als Demo wird das Datenbyte 'data1' in die Adresse
; 'byte_adr' geschrieben.
; Durch Acknowledge-Polling wird das Ende der Programmierung
; abgefragt. (Achtung: Es werden nur die unteren 256 Bytes des
; EEPROMS angesprochen. Die oberen drei Bits B2, B1 und B0
; werden immer auf low gesetzt!)
; AT90S2313 mit 4 MHz Takt
;***************************************************************

.device AT90S2313
.include „2313def.inc"



;***************************************************************
; Konstantendefinition
;***************************************************************

; Schnittstelle PIC <-> EEPROM
.equ        SCL     = PB0           ;Serial Clock (PB0)
.equ        SDA_out = PB1           ;Serial Data Out (PB1)
.equ        SDA_in  = PB2           ;Serial Data In (PB2)

; andere Konstanten
.equ        byte_adr    = 01h       ;Byte-Adresse
.equ        dev_read    = 0b10100001;Device-Select R/W=1
.equ        dev_write   = 0b10100000;Device-Select R/W=0
.equ        data1       = 0xAA      ;Datenbyte = AAh
.equ        bit_anz     = 0x08      ;1 Byte = 8 Bits



;***************************************************************
; Variablendefinition
;***************************************************************

.def        buffer  = r16           ;Buffer, Register r16
.def        counter = r17           ;Zaehler, Register r17
.def        temp    = r18
```

```
;*************************************************************
; Programm
;*************************************************************
; Das Programm beginnt bei Adresse 0x00

.CSEG
.ORG        0x00                    ; Programm beginnt bei 0
            rjmp    main            ; Starte Hauptprogramm


;*************************************************************
; Subroutine 'init'
; Die Routine initialisiert den I2C-Bus
; Die Pins PB0 (SCL) und PB1 (SDA_out) sind Ausgaenge
; Der Pin PB2 (SDA_in) ist ein Eingang
; Die Leitungen SCL und SDA_out werden losgelassen (high)
;
;*************************************************************

init:       sbi     PORTB,SDA_out   ; SDA high
            sbi     PORTB,SCL       ; SCL high
            ldi     temp,0b11111011 ; PB0 und PB1 Ausgang,
            out     DDRB,temp       ; PB2 Eingang
            ret


;*************************************************************
; Subroutine 'pstart'
; Diese Routine sendet eine Startbedingung
; SCL=high (PB0)
; SDA_out=high-low-Flanke (PB1)
;
; Nach der Startbedingung ist SCL auf low
;*************************************************************

pstart:     sbi     PORTB,SCL       ;SCL auf high
            nop                     ;warten
            nop
            cbi     PORTB,SDA_out   ;SDA auf low waehrend SCL high
            nop                     ;warten
            nop
            cbi     PORTB,SCL       ;SCL auf low
            sbi     PORTB,SDA_out   ;SDA loslassen
            ret
```



```
;*************************************************************
; Subroutine 'pstop'
; Diese Routine sendet eine Stopbedingung
; SCL=high (PRB0)
; SDA_out=low-high-Flanke (PB1)
;
; Nach der Stopbedingung sind SCL und SDA auf high
;*************************************************************

pstop:      cbi     PORTB,SDA_out   ;SDA auf low zurueck
            nop                     ;warten
            nop
            sbi     PORTB,SCL       ;SCL auf high
            nop                     ;warten
            nop
            sbi     PORTB,SDA_out   ;SDA auf high wahrend SCL high
            ret


;*************************************************************
; Subroutine 'rx_ack'
; Diese Routine prueft, ob EEPROM das ACK-Bit sendet
;
; Kehrt mit SCL low zurueck
;*************************************************************

rx_ack:     sbi     PORTB,SCL       ;erzeuge Takt
            nop                     ;warte
            nop
warte:      sbic    PINB,SDA_in     ;Teste auf ACK
            rjmp    warte           ;ACK ist low, warte
            cbi     PORTB,SCL       ;SCL wieder low
            ret


;*************************************************************
; Subroutine 'tx_ack'
; Diese Routine sendet ein ACK-Bit=0
;
; Kehrt mit SCL low zurueck
;*************************************************************

tx_ack:     cbi     PORTB,SCL       ;Takt vorbereiten
            cbi     PORTB,SDA_out   ;SDA auf low
            sbi     PORTB,SCL       ;Takt erzeugen
```

```
            nop                     ;warten
            cbi     PORTB,SCL
            sbi     PORTB,SDA_out   ;SDA loslassen
            ret



;*****************************************************************
; Subroutine 'tx_no_ack'
; Diese Routine sendet ein ACK-Bit=1
;
; Kehrt mit SCL low zurueck
;*****************************************************************

tx_no_ack:  cbi     PORTB,SCL       ;Takt vorbereiten
            sbi     PORTB,SDA_out   ;SDA auf high
            sbi     PORTB,SCL       ;Takt erzeugen
            nop                     ;warten
            cbi     PORTB,SCL
            ret



;*****************************************************************
; Subroutine 'ack_poll'
; Diese Routine fuehrt ein Acknowledge-Polling aus.
;
; Bei der weiteren Ausfuehrung ist zu beachten, dass diese
; Routine bereits ein Device-Select-Byte mit R/W=0 sendet.
; Kehrt mit SCL low zurueck.
;*****************************************************************

ack_poll:   rcall   pstart          ;Startbegingung senden
            ldi     buffer,dev_write
            rcall   byte_out
            sbi     PORTB,SCL       ;erzeuge Takt
            nop                     ;warte
            nop
            sbic    PINB,SDA_in     ;Teste auf ACK
            rjmp    ack_poll
            cbi     PORTB,SCL       ;SCL wieder low
            ret
```



```
;*****************************************************************
; Subroutine 'byte_out'
; Diese Routine sendet ein Byte
; Das Byte muss sich im Register „buffer" befinden
;
; SCL befindet sich anfangs auf low (nach START)
; SDA befindet sich anfangs auf high (nach START)
;*****************************************************************

byte_out:   ldi     counter,bit_anz ;8 Bits uebertragen

next_out:   cbi     PORTB,SCL       ;SCL low (Vorbereitung)
            sbi     PORTB,SDA_out   ;SDA high (Vorbereitung)
            rol     buffer          ;schiebe MSB ins Carry
            brcs    wr_one          ;falls Bit high, alles o.k.
            cbi     PORTB,SDA_out   ;Nein! Setze SDA auf low
wr_one:     sbi     PORTB,SCL       ;erzeuge Takt
            dec     counter         ;alle Bits uebertragen?
            brne    next_out        ;Nein! Naechstes Bit
            cbi     PORTB,SCL       ;SCL wieder auf low
            nop
            nop
            sbi     PORTB,SDA_out   ;SDA wieder loslassen
            ret



;*****************************************************************
; Subroutine 'byte_in'
; Diese Routine liest ein Byte vom EEPROM. Das gelesene Byte
; wird in der Variablen 'buffer' uebergeben.
;*****************************************************************

byte_in:    ldi     counter,bit_anz ;es werden 8 Bits uebertragen

next_in:    clc                     ;Carry loeschen
            sbi     PORTB,SCL       ;Takt erzeugen
            nop                     ;warten
            nop
            sbic    PINB,SDA_in     ;hat EEPROM eine „0" gesendet?
            sec                     ;Nein! Setze Carry selber
            rol     buffer          ;schiebe gelesenes Bit in Buffer
            cbi     PORTB,SCL       ;SCL wieder low
            dec     counter         ;Alle 8 Bits gelesen?
            brne    next_in         ; Nein! Lies naechstes Bit
            ret                     ;Ja! Springe zurueck
```

```
;***************************************************************
; Hauptprogramm 'main'
;
;***************************************************************

main:       ldi     temp,RAMEND     ; setze Stack-Pointer
            out     SPL,temp        ; an das SRAM-Ende
            rcall   init            ; Schnittstelle initialisieren

            ;Ein Byte Schreiben
            rcall   pstart          ;Start senden

            ldi     buffer,dev_write;Steuerwort senden (R/W=0)
            rcall   byte_out
            rcall   rx_ack          ;auf ACK-Bit testen

            ldi     buffer,byte_adr ;Byte-Adresse senden
            rcall   byte_out
            rcall   rx_ack          ;auf ACK-Bit testen

            ldi     buffer,data1    ;Daten senden
            rcall   byte_out
            rcall   rx_ack          ;auf ACK-Bit testen
            rcall   pstop           ;Stop senden

            ;Acknowledge-Polling ausfuehren
            rcall   ack_poll

            ;Byte lesen
            ;Start und Device-Select bereits in ack_poll ausgefuehrt
            ldi     buffer,byte_adr ;Byte-Adresse senden
            rcall   byte_out        ;(nur untere 8 Bits!)
            rcall   rx_ack

            rcall   pstart

            ldi     buffer,dev_read ;Steuerwort senden (R/W=1)
            rcall   byte_out
            rcall   rx_ack
            rcall   byte_in         ;Adressiertes Byte von EEPROM
                                    ;Empfangen
            rcall   tx_no_ack       ;ACK=1 senden
            rcall   pstop           ;Uebertragung beenden

wieder:     rjmp    wieder          ;Endlosschleife
```


5.3 Ansteuerung einer LCD-Anzeige

In diesem Abschnitt wird die Ansteuerung von LCD-Anzeigen besprochen, die auf dem LCD-Controller HD44780 von Hitachi oder einem kompatiblen Typ basieren. Solche Anzeigen gibt es in den unterschiedlichen Ausführungen, so z. B. ein- bis vierzeilig mit 8 bis 40 Zeichen pro Zeile. In diesem Beispiel wird eine Anzeige mit zwei Zeilen und 16 Zeichen pro Zeile verwendet. Alle Überlegungen können aber fast ausnahmslos auf die anderen Anzeigetypen übertragen werden, da der interne Aufbau der Anzeigen sich nur minimal – in der Anzahl der zusätzlich benötigten LCD-Treiber Chips – unterscheidet. Das Anzeigemodul kommt mit einer einzigen Versorgungsspannung von 5 V aus. Der Anschluß an einen Mikrocontroller ist sehr einfach und erfolgt entweder über einen 4 oder 8 Bit breiten Bus. Möchte man Portleitungen sparen, wählt man einen 4 Bit breiten Bus, muß dann aber die Daten und Befehle in zwei Gruppen von je 4 Bit aufspalten und hintereinander übertragen. Die Ersparnis an Leitungen wird dann durch einen umfangreicheren Programmcode erkauft. Zur besseren Übersicht wird hier die 8 Bit breite Übertragung verwendet werden. Neben den Datenleitungen werden noch drei Steuerleitungen benötigt, um die Anzeige anzusprechen. Die Anschlußbelegung des verwendeten Anzeigemoduls ist der **Tabelle 5.9** zu entnehmen.

Pin-Nummer	Bezeichnung	Beschreibung
1	V_{SS}	Masse
2	V_{CC}	Versorgungsspannung +5V
3	V_O	Kontrast
4	RS	Register Select
5	R/W	Read/Write
6	E	Enable
7	DB0	Datenleitung
.	.	.
.	.	.
.	.	.
14	DB7	Datenleitung

Tabelle 5.9: Die Anschlußbelegung des verwendeten Anzeigemoduls.

Die Spannung am Eingang V_O stellt den Kontrast der Anzeige ein. Üblicherweise wird die Spannung über einen Spannungsteiler diesem Eingang zugeführt (siehe Schaltplan). Die drei Steuerleitungen regeln den Datenverkehr von und zur Anzeige (siehe **Tabelle 5.10**).

Tabelle 5.10: Funktion der Steuerleitungen E, R/W und RS.

Steuerleitung	logischer Pegel	Funktion
E	0	Anzeige deaktiviert
	1	Anzeige aktiviert
R/W	0	Schreibt Daten zur Anzeige
	1	Liest Daten von der Anzeige
RS	0	Daten als Befehl interpretieren
	1	Daten als Zeichen interpretieren

Die Steuerleitung E (Enable) aktiviert/deaktiviert die Anzeige. Ist die Anzeige aktiviert, überprüft diese die Zustände der beiden anderen Steuerleitungen und wertet danach die Datenleitungen entsprechend aus. Wird die Anzeige deaktiviert, werden die Zustände der anderen Steuerleitungen ignoriert und die Datenleitungen werden hochohmig (Tri-State) geschaltet. Der Datenbus kann dann für andere Zwecke verwendet werden. Die Steuerleitung R/W (Read/Write) signalisiert, ob Daten zur Anzeige geschrieben werden oder von dieser gelesen werden sollen. Schließlich gibt die Leitung RS (Register Select) an, ob es sich bei den übertragenen Daten um Befehle für den Anzeigencontroller oder um Zeichen, die in die Anzeige geschrieben werden sollen, handelt.

Unabhängig von der Anzahl der Zeilen und Zeichen pro Zeile, verfügt der Anzeigencontroller HD44780 über 80 Byte internes RAM, auch Data Display (DD) RAM genannt, das zur Darstellung der Zeichen in der Anzeige dient. Bei einer zweizeiligen Anzeige mit 16 Zeichen pro Zeile ist dem ersten Zeichen (das äußerst linke Zeichen in der 1. Zeile) die RAM-Adresse 00h zugeordnet. Die Adressierung erfolgt in der ersten Zeile dann weiter fortlaufend bis zur 40. Position,

die der RAM-Adresse 27h zugeordnet ist. Die Zeichen der zweiten Zeile werden über die Adressen 40h bis 67h angesprochen. Möchte man also z. B. ein Zeichen an die dritte Position (von links gezählt) der zweiten Zeile schreiben, so muß man das Zeichen in die DD-RAM Adresse 42h schreiben. Von den zweimal 40 Bytes DD-RAM sind natürlich immer nur zweimal 16 gleichzeitig sichtbar. Stellt man sich den DD-RAM Bereich zweizeilig mit je 40 Stellen vor, so sieht man immer nur ein Fenster aus zweimal 16 Zeichen. Der HD44780 unterstützt Befehle (**Tabelle 5.11**), die dieses Fenster nach links und rechts verschieben können, um so andere Bereiche des DD-RAMs sichtbar zu machen (**Bild 5.19**).

Bild 5.19: DD-RAM Bereich und Zeichenposition bei einer zweizeiligen Anzeige mit 16 Zeichen pro Zeile.

Neben dem DD-RAM verfügt der HD44780 noch über einen Character Generator ROM (CG-ROM) und CG-RAM. Im CG-ROM (Zeichensatzgenerator) befinden sich die festprogrammierten, darstellbaren Zeichen, während im CG-RAM acht zusätzliche 5 x 7 beziehungsweise vier 5 x 10 Punkte Zeichen vom Anwender generiert werden können (**Bild 5.20**).

Bei der Programmierung eines Anzeigemoduls geht man folgendermaßen vor: Nach dem Einschalten läuft eine interne Rücksetzsequenz ab. Während dieser Zeit, die typisch 10 ms beträgt, hält der HD44780 das Busy-Flag auf logisch Eins. Dieses Busy-Flag signalisiert auch bei der Ausführung anderer Befehle, daß der Controller beschäftigt ist und keine weiteren Daten verarbeiten kann. Grundsätzlich sollte man, bevor man Daten zum Controller schickt, den Zustand des Busy-Flags mit dem entsprechenden Befehl aus **Tabelle 5.11** abfragen.

Instruction	Code										Description	Execution time (when f(cp) or f(OSC) = 250 kHz) Note 1	Execution time (when f(cp) or f(OSC) = 190 kHz) Note 2
	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0			
Clear display	0	0	0	0	0	0	0	0	0	1	Clears all display and returns the cursor to the home position (Address 0).	82 μs – 1.64 ms	120 μs – 4.8 ms
Return home	0	0	0	0	0	0	0	0	1	*	Returns the cursor to the home position (Address 0). Also returns the display being shifted to the original position. DD RAM contents remain unchanged.	40 μs – 1.6 ms	120 μs – 4.8 ms
Entry mode set	0	0	0	0	0	0	0	1	I/D	S	Sets the cursor move direction and specifies or not to shift the display. These operations are performed during data write and read.	40 μs	120 μs
Display ON/OFF control	0	0	0	0	0	0	1	D	C	B	Sets ON/OFF of all display (D), cursor ON/OFF (C), and blink of cursor position character (B).	40 μs	120 μs
Cursor and display shift	0	0	0	0	0	1	S/C	R/L	*	*	Moves the cursor and shifts the display without changing DD RAM contents.	40 μs	120 μs
Function set	0	0	0	0	1	DL	N	F	*	*	Sets interface data length (DL), number of display lines (L) and character fonts (F).	40 μs	120 μs
Set CG RAM address	0	0	0	1	A_{CG}						Sets the CG RAM address. CG RAM data is sent and received after this setting.	40 μs	120 μs
Set DD RAM address	0	0	1	A_{DD}							Sets the DD RAM address. DD RAM data is sent and received after this setting.	40 μs	120 μs
Read busy flag & address	0	1	BF	AC							Reads Busy Flag (BF) indicating internal operation is being performed and reads address counter contents.	1 μs	1 μs
Write data to CG or DD RAM				Write data							Writes data into DD RAM or CG RAM.		
Read data to CG or DD RAM	1	1		Read Data							Reads data from DD RAM or CG RAM.	40 μs	120 μs
	I/D = 1: Increment (+1) I/D = 0: Decrement (–1) S = 1: Accompanies display shift. S/C = 1: Display shift S/C = 0: Cursor move R/L = 1: Shift to the right. R/L = 0: Shift to the left. DL = 1: 8 bits DL = 0: 4 bits N = 1: 2 lines N = 0: 1 line F = 1: 5 x 10 dots F = 0: 5 x 7 dots BF = 1: Internally operating BF = 0: Can accept instruction										DD RAM: Display data RAM CG RAM: Character generator RAM A_{CG}: CG RAM address A_{DD}: DD RAM address. Corresponds to cursor address. AC: Address counter used for both of DD and CG RAM address.	Execution time changes when frequency changes. (Example) When f(cp) or f(OSC) is 270 kHz: $40\ \mu s \times \frac{250}{270} = 37\ \mu s$	

Tabelle 5.11: Befehle und deren Funktion des LCD-Controllers HD44780.

Bild 5. 20: Der Zeichensatz des LCD-Controllers HD44780.

Der Befehl zur Abfrage des Busy-Flags ist der einzige, der ausgeführt wird, auch wenn der Controller beschäftigt ist. Die Zeitdiagramme zum Schreiben und Lesen sind in den **Bildern 5.21** und **5.22** dargestellt.

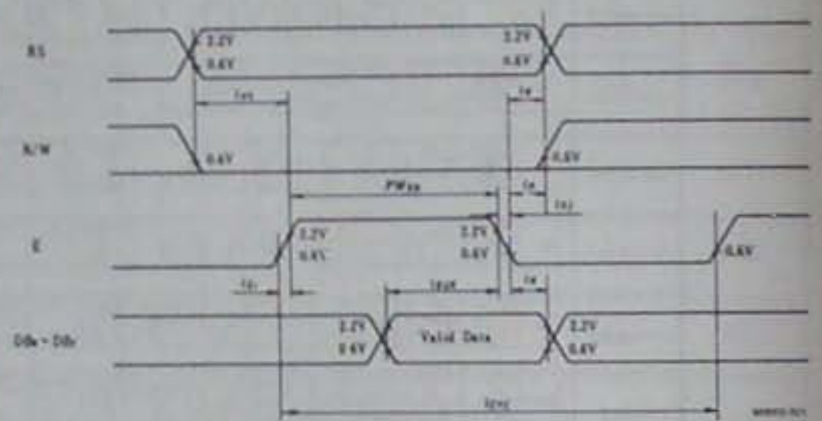


Bild 5.21: Daten zur Anzeige senden.

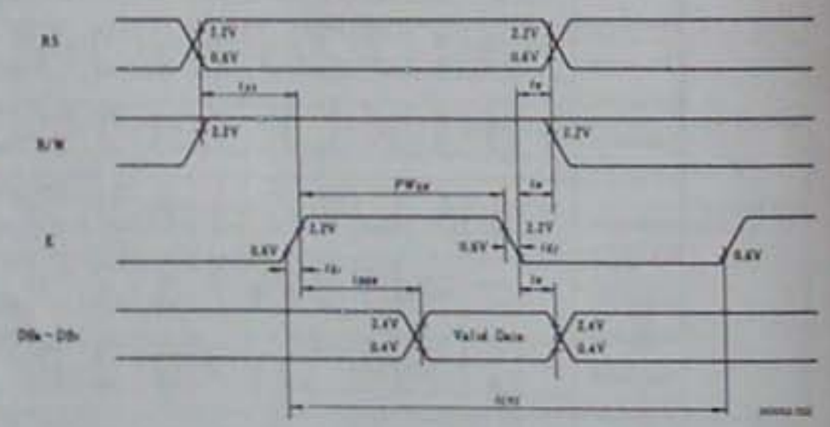


Bild 5.22: Daten von der Anzeige empfangen.

Symbol	Min.	Max.	Einheit
t_{CYC}	1,0	–	µs
P_{WEH}	450	–	ns
$t_{R,TF}$	–	25	ns
t_{AS}	140	–	ns
t_{DDR}	–	320	ns
t_{DSW}	195	–	ns
t_{H}	20	–	ns

Tabelle 5.12: Zeitangaben.

Der Anschluß des Anzeigemoduls an die Experimentierplatine erfordert lediglich einen Trimmer, mit dem man den Kontrast der Anzeige an die Umgebungshelligkeit anpassen kann. Die Datenleitungen werden am Port B des AT90S2313 angeschlossen. Über drei Leitungen des Ports D wird die Anzeige gesteuert (**Bild 5.23**).

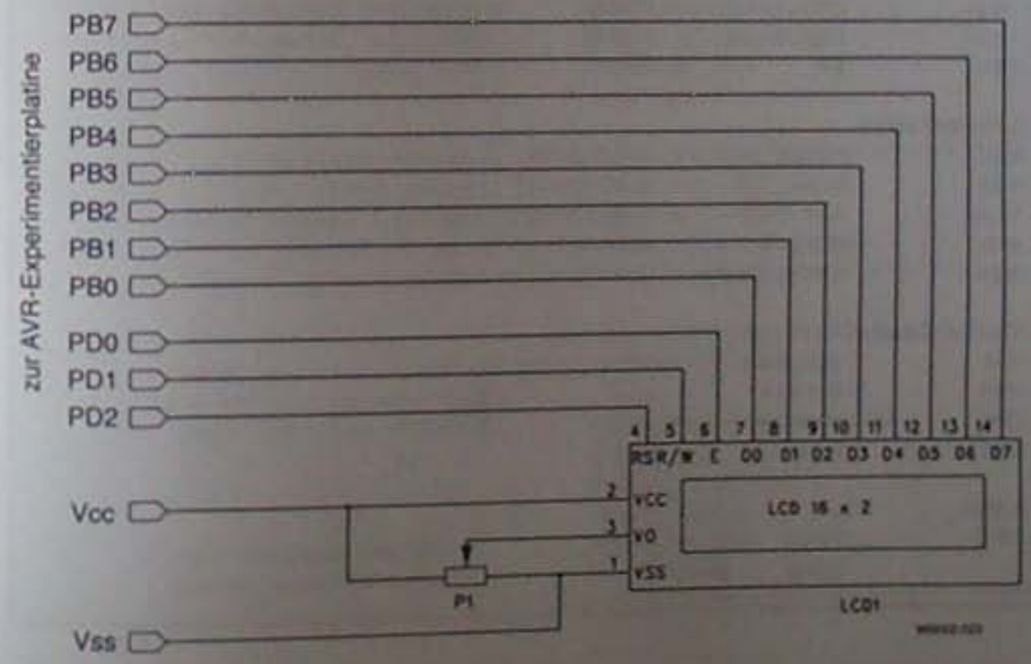


Bild 5.23: Schaltplan zum Anschluß des Anzeigemoduls an die AVR-Experimentierplatine.

Das Beispielprogramm liest Daten aus Tabellen und sendet diese in die erste und zweite Zeile der Anzeige. Die einzelnen Programmteile sind so ausgelegt, daß diese sehr einfach in eigene Programme übernommen werden können.

```
;***************************************************************
;Programm LCD.ASM
;Das Programm demonstriert, wie man eine LCD-Anzeige mit
;16 Zeichen x 2 Zeilen an einen AT90S2313 anschliesst.
;Es wird vorausgesetzt, dass die Anzeige einen HD44780 oder
;kompatiblen Controller verwendet.
;AT90S2313 mit 4 MHz Takt
;***************************************************************

.device AT90S2313
.include „2313def.inc"                ;muss im selben Verzeichnis stehen


;Definition der Steuerleitungen
.equ        E           = PD0         ;E an Pin PD0
.equ        RW          = PD1         ;RW an Pin PD1
.equ        RS          = PD2         ;RS an Pin PD2
;.equ       LCD_cntr    = PORTD       ;PORTD ist Kontroll-Port
;.equ       LCD_data    = PORTB       ;PORTB ist Daten-Port
.equ        BF          = PB7         ;Busy flag

;LCD Befehle
.equ        clear_LCD   = 0b00000001 ;loesche Anzeige
.equ        home_LCD    = 0b00000010 ;return home
.equ        set_LCD     = 0b00111000 ;8 bits,2 Zeilen,5x7dots
.equ        LCD_on      = 0b00001110 ;schalte LCD ein
.equ        entry_mode  = 0b00000110 ;setze Cursor

;Variablendefinition
.def        zeichen     = r0          ;Zeichen aus der Tabelle
.def        buffer      = r16         ;RX/TX Daten von/zu LCD
.def        counter     = r17         ;Zaehler fuer den Text
.def        temp        = r18

.CSEG
.ORG        0x00                      ; Programm beginnt bei 0
            rjmp    main              ; Starte Hauptprogramm
```



```
;***************************************************************
; Subroutine init
; Initialisiere PORTD
;***************************************************************

init:       ldi     temp,0b11111111 ;PORTD ist Ausgang
            out     DDRD,temp
            cbi     PORTD,E         ;E initialisieren
            cbi     PORTD,RS
            cbi     PORTD,RW
            ret

;***************************************************************
; Subroutine busy_flag
; Diese Routine testet, ob die LCD-Anzeige bereit ist, einen
; neuen Befehl oder weitere Daten zu empfangen.
;***************************************************************

busy_flag:  ldi     temp,0b00000000 ;PORTB ist Eingang
            out     DDRB,temp
            cbi     PORTD,RS        ;Befehl wird gesendet
            sbi     PORTD,RW        ;setze LCD in Lesemodus
            sbi     PORTD,E         ;spreche LCD an
            nop
            nop
            sbic    PINB,BF         ;LCD bereit?
            rjmp    busy_flag       ;nein, wiederhole
            cbi     PORTD,E         ;disable LCD
            ret                     ;LCD bereit

;***************************************************************
; Subroutine write_data
; Diese Routine sendet Daten zur LCD-Anzeige.
; Die Daten muessen im Register buffer uebergeben werden.
;***************************************************************

write_data: rcall   busy_flag       ;LCD bereit?
            ldi     temp,0b11111111 ;PORTB ist Ausgang
            out     DDRB,temp
            sbi     PORTD,RS        ;Daten werden gesendet
            cbi     PORTD,RW        ;LCD in Schreibmodus
            sbi     PORTD,E         ;spreche LCD an
            out     PORTB,buffer    ;sende Daten
            cbi     PORTD,E         ;disable LCD
            ret
```

```
;***********************************************************
; Subroutine write_instr
; Diese Routine sendet Befehle zur LCD-Anzeige.
; Der Befehl muss im Register buffer uebergeben werden.
;***********************************************************

write_instr: rcall busy_flag        ;LCD bereit?
             ldi   temp,0b11111111  ;RB ist Ausgang
             out   DDRB,temp
             cbi   PORTD,RS         ;Befehl wird gesendet
             cbi   PORTD,RW         ;LCD in Schreibmodus
             sbi   PORTD,E          ;spreche LCD an
             out   PORTB,buffer     ;sende Befehl
             cbi   PORTD,E          ;disable LCD
             ret

;***********************************************************
; Hauptprogramm
; Schreibt „AT90S2313" in die erste Zeile und „LCD-Routine"
; in die zweite Zeile des LCDs.
;***********************************************************

main:        ldi   temp,RAMEND      ; setze Stack-Pointer
             out   SPL,temp         ; an das SRAM-Ende

             rcall init             ; PORTD initialisieren

             ldi   buffer,set_LCD   ;setze LCD Funktion
             rcall write_instr

             ldi   buffer,LCD_on    ;schalte LCD ein
             rcall write_instr

             ldi   buffer,clear_LCD ;loesche Anzeige
             rcall write_instr

             ldi   buffer,entry_mode ;Eingabemodus
             rcall write_instr

;Holt den Text 'AT90S2313' aus der Tabelle und schreibt diesen in
;die erste Zeile der Anzeige

             ldi   counter,9        ;Zeichenzaehler
             ldi   ZL,LOW(Tabelle1*2) ;Low-Zeiger auf Tabellenanfang
             ldi   ZH,HIGH(Tabelle1*2);High-Zeiger auf Tabellenanfang
```



```
loop_msg1:   lpm                    ;hole Zeichen aus Tabelle
             mov   buffer,zeichen   ;Zeichen uebergeben
             rcall write_data       ;schreibe Zeichen in LCD
             inc   ZL               ;Low-Zeiger um 1 erhoehen
             brcc  no_carry1        ;kein Uebertrag von ZL
             inc   ZH               ;ZH erhoehen, da Uebertrag von ZL
no_carry1:   dec   counter          ;alle Zeichen gesendet?
             brne  loop_msg1        ;Nein! Sende naechstes Zeichen

;Holt den Text 'LCD-Routine' aus der Tabelle und schreibt diesen
;in die zweite Zeile der Anzeige

             ldi   buffer,0b11000000;LCD-Startadresse 2.Zeile
             rcall write_instr      ;sende Befehl
             ldi   counter,11       ;Zeichenzaehler
             ldi   ZL,LOW(Tabelle2*2);Low-Zeiger auf Tabellenanfang
             ldi   ZH,HIGH(Tabelle2*2);High-Zeiger auf Tabellenanfang
loop_msg2:   lpm                    ;hole Zeichen aus Tabelle
             mov   buffer,zeichen   ;Zeichen uebergeben
             rcall write_data       ;schreibe Zeichen in LCD
             adiw  ZL,1             ;16-Bit Pointer um 1 erhöhen
no_carry2:   dec   counter          ;alle Zeichen gesendet?
             brne  loop_msg2        ;Nein! Sende naechstes Zeichen

;Endlosschleife, AVR zuruecksetzen, um Programm erneut zu starten

loop:        rjmp  loop

;Tabelle 1 mit dem Text „AT90S2313", der in die 1.Zeile geschrieben
;werden soll

Tabelle1:    .DB „AT90S2313"

;Tabelle 2 mit dem Text „LCD-Routine", der in die 2.Zeile
;geschrieben werden soll

Tabelle2:    .DB „LCD-Routine"
```

5.4 Telefonkartenleser

Die Funktion und der Aufbau der Guthabenkarte für öffentliche Kartentelefone, im folgenden einfach als Telefonkarte bezeichnet, sind bereits in zahlreichen Publikationen offengelegt worden. Auf der Telefonkarte befindet sich ein Chip mit insgesamt 104 Bit Speichervolumen. Bei dem verwendeten Chiptyp handelt es sich um einen einstellbaren Zähler, der nach Herunterzählen (Verbrauchen der Einheiten) nicht mehr zurücksetzbar ist. Die äußeren Maße einer Telefonkarte entsprechen denen einer Kredit- oder auch EC-Karte. Der Chip verbirgt sich hinter acht Kontakten, dem sogenannten Chipmodul, auf der Karte. Die Kontaktlage und -bezeichnung kann man aus **Bild 5.24** entnehmen.

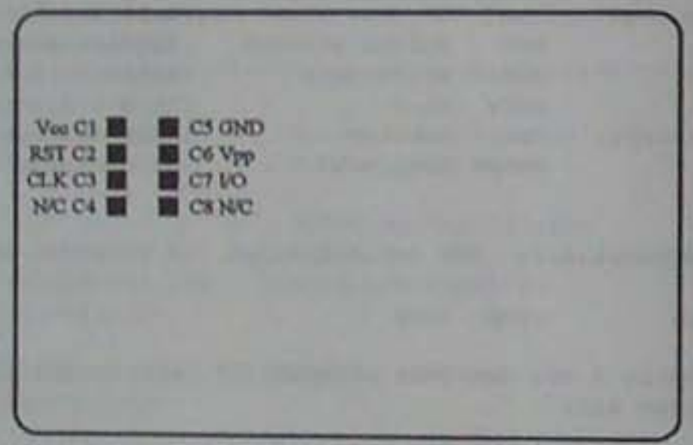


Bild 5.24: Kontaktlage und -bezeichnung der Telefonkarte.

Die acht Kontakte der Chipkarten werden mit C1 bis C8 durchnummeriert. Diese sind: Spannungsversorgung (C1: V_{CC}), Reset (C2: RST), Takt (C3: CLK), Masse (C5: GND), Programmierspannung (C6: V_{pp}) – die bei heutigen Chipkarten nicht mehr benötigt wird, da die Programmierspannung durch Ladungspumpen auf den Chip erzeugt wird – und Dateneingang/-ausgang (C7: I/O). Die Kontakte C4 und C8 sind für zukünftige Anwendungen reserviert und nicht angeschlossen.

Beim Lesen verhält sich ein Telefonkartenchip wie ein ganz normales 104-Bit-EEPROM. Eine integrierte Logik liefert die gespeicherten Bits nach einem Lesebefehl nacheinander an die I/O-Leitung. Der eigentliche Lesebefehl besteht aus einem kurzen Reset-Impuls, worauf bei jedem weiteren Impuls auf der Taktleitung das jeweils nächste Bit auf der I/O-Leitung anliegt (**Bild 5.25**).

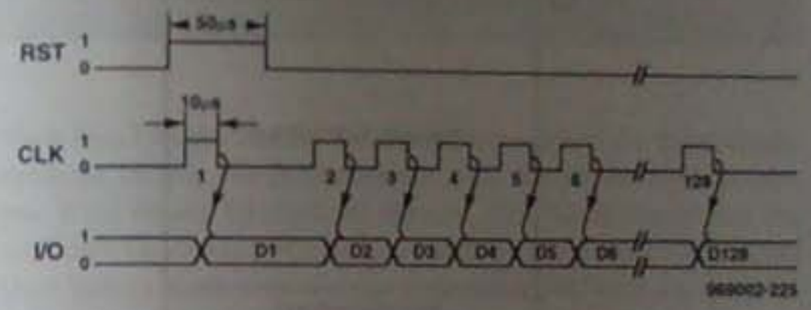


Bild 5.25: Signalverläufe zur Ansteuerung einer synchronen Karte nach ISO 7816-3.

Die 104 Bits gliedern sich in:

- 16-Bit-ROM für nicht veränderbare Daten, wie Herstellercodierung
- 8-Bit blockierbares EEPROM für Herstellerdaten, die in der Testphase vom Chiphersteller programmiert werden
- 40-Bit blockierbares PROM für Personalisierdaten
- 8-Bit-PROM inklusive Personalisierbit
- 32-Bit-EEPROM, als Oktalzähler konfiguriert

Die fünfstellige Real-Seriennummer, Prüfwert, Herstellungsmonat und -jahr, Herstellercode sowie der Kartencode (dabei handelt es sich um das Anfangsguthaben) sind binär codiert in zehn Nibbles (Halbbytes) im 40-Bit-ROM abgelegt.

Bild 5.26:
Speichermap der Telefonkarte.

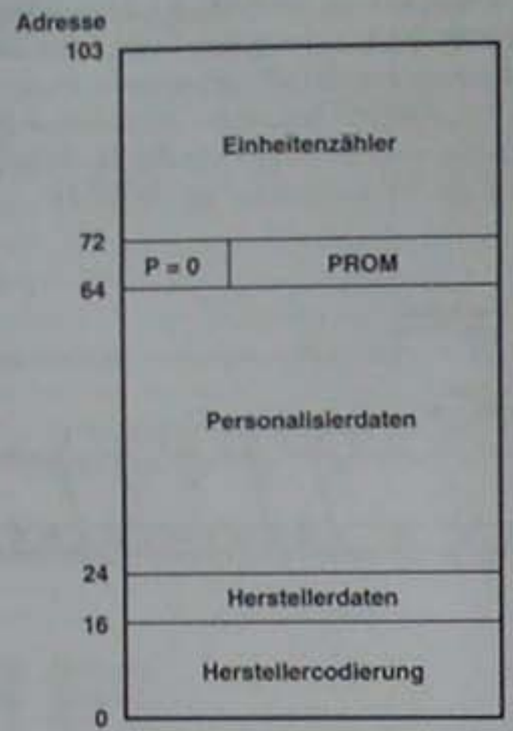


Tabelle 5.13:
Codierung der Hersteller.

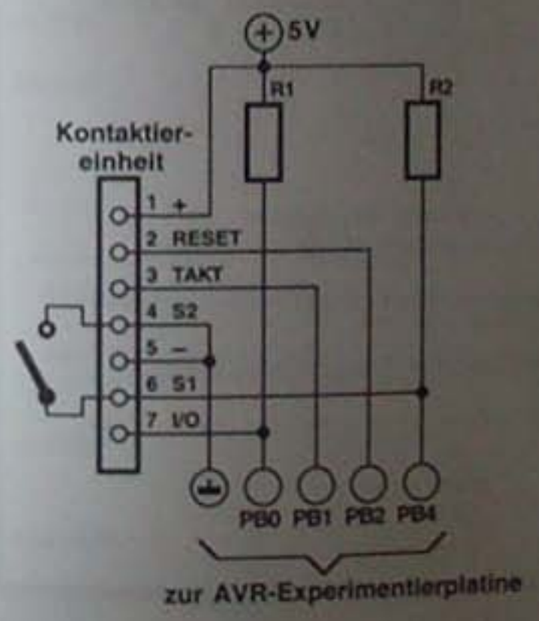
Herstellercode	Hersteller
0	Orga Kartensysteme
1	Giesecke & Devrient
2	Oldenburg Datensysteme
3	Gemplus
4	Solaic
5	Uniqua
6	Schlumberger

Kartencode	Anfangsguthaben
3	1,50 DM
4	6,00 DM
5	12,00 DM
6	50,00 DM

Tabelle 5.14:
Codierung des Anfangsguthabens.

Nach dem Einschieben in ein Kartentelefon liefert die Telefonkarte zunächst 16 Bits, die Informationen über die Art der Karte enthalten. Wird sie als Telefonkarte erkannt, liest das Kartentelefon die restlichen Daten aus und schickt sie über ein sogenanntes DOV (Data Over Voice)-Modem an den Zentralrechner der Telekom, der wiederum gesperrte Karten herausfinden und sperren kann.

Wie die Experimentierplatine zu einem komfortablen Telefonkartenleser erweitert werden kann, ist in **Bild 5.27** zu sehen.



Stückliste:

Widerstände:
R1, R2 = 10 kΩ

Außerdem:
Kontaktiereinheit
ITT-Cannon,
CCM02-2NO-32

Bild 5.27:
Schaltplan zum Anschluß einer Telefonkartenleseeinheit an die AVR-Experimentierplatine.

Tabelle 5.15:
Stückliste Telefonkartenleseeinheit.

Das im folgenden abgedruckte Programm stellt die Grundroutinen zum Lesen einer Telefonkarte bereit. Der Leser kann dadurch sehr einfach das Programm für eigene Projekte einsetzen und bei Bedarf modifizieren.

```
;***********************************************************
; Telefonkartenleser
; Version 0.99
;
; File-Name: TKL.ASM
; AT90S2313 mit 4 MHz Takt
;***********************************************************

.device AT90S2313
.include „2313def.inc"

;Konstanten fuer Telefonkarten Interface
.equ        I_O     = PB0           ; RB.0 von Port B ist I/O
.equ        CLK     = PB1           ; RB.1 von Port B ist CLK
.equ        RST     = PB2           ; RB.2 von Port B ist RST
.equ        S1      = PB4           ; RB.4 von Port B ist S1


;Variablen
.def        delay_cntr = r16        ; Schleifenzaehler in FR 8
.def        buffer     = r17        ; Datenbyte in FR 9
.def        counter    = r18        ; Zaehler
.def        temp       = r19


.CSEG
.ORG        0x00                    ; Programm beginnt bei 0
            rjmp    main            ; Starte Hauptprogramm

;***********************************************************
; Soubroutine rst_tk
; Diese Subroutine gibt einen Reset an die Telefonkarte aus
; und liefert das erste Bit.
;***********************************************************

rst_tk:     sbi     PORTB,RST       ;RST high
            ldi     counter,25      ;warte
schleife1:  dec     counter         ;Zaehler
```



```
            brne    schleife1
            rcall   clock           ;sende Takt
            ldi     counter,25      ;warte
schleife2:  dec     counter         ;Zaehler
            brne    schleife2
            cbi     PORTB,RST       ;RST low
            ret                     ;Ruecksprung


;***********************************************************
; Soubroutine clock_N
; Gibt N Takimpulse aus. Die Anzahl N muss im Register
; „counter" uebergeben werden.
;***********************************************************

clock_N:    rcall   clock           ;Takt ausgeben
            dec     counter         ;Zaehler
            brne    clock_N
            ret                     ;Ruecksprung


;***********************************************************
; Soubroutine clock
; Erzeugt einen Taktimpuls
;***********************************************************

clock:      sbi     PORTB,CLK       ;CLK high
            ldi     delay_cntr,14   ;warte
loop2:      dec     delay_cntr      ;Zaehler
            brne    loop2
            cbi     PORTB,CLK       ;CLK low
            ret                     ;Rueckspung


;***********************************************************
; Soubroutine read4
; Liest vier Bits ein und legt diese ins untere Nibble
; der Variablen 'buffer'.
;***********************************************************

read4:      clr     buffer          ;bereite buffer vor
            clc                     ;bereite Carry vor
            ldi     counter,4       ;lies 4 Bits
weiter4:    rcall   clock           ;sende Takt
            sbic    PINB,I_O        ;Bit1 empfangen?
```

```
            sec                     ;NEIN! Eins empfangen
            ror     buffer          ;sichere gelesenes Bit
            dec     counter         ;4 Bits gelesen?
            brne    weiter4         ;Nein! Lies weiter
            swap    buffer          ;vertausche Nibbles
            ret


;***************************************************************
; Soubroutine read8
; Liest acht Bits ein und legt diese in die Variable 'buffer'.
;***************************************************************

read8:      clr     buffer          ;bereite buffer vor
            clc                     ;bereite Carry vor
            ldi     counter,8       ;lies 8 Bits
weiter8:    rcall   clock           ;sende Takt
            sbic    PINB,I_O        ;Null empfangen?
            sec                     ;NEIN! Eins empfangen
            ror     buffer          ;sichere gelesenes Bit
            dec     counter         ;8 Bits gelesen?
            brne    weiter8         ;Nein! Lies weiter
            ret


;***************************************************************
;Hauptprogramm
;
; Als Beispiel wird folgendes ausgefuehrt:
; 1. Karte eingesteckt?
; 2. Reset senden
; 3. Adresszaehler auf Adresse 24 setzen
; 4. Herstellercode lesen
; 5. Adresszaehler auf Adresse 32 setzen
; 6. Kartencode lesen
; 7. Jahr und Monat lesen
; 9. Karte noch eingesteckt?
;
; Die Daten werden in ein Register geladen und nicht weiter
; verarbeitet. Der interessierte Leser kann das Programm sehr
; einfach beliebig ergaenzen.
;***************************************************************

main:       ldi     temp,RAMEND     ;setze Stack-Pointer
            out     SPL,temp        ;an das SRAM-Ende
```



```
            ldi     temp,0b00001110 ;PB0,PB4: Eingang
            out     DDRB,temp       ;PB1,PB2: Ausgang
            cbi     PORTB,RST       ;RST auf low

no_tk:      sbic    PINB,SI         ;Ist Karte eingesteckt?
            rjmp    no_tk           ;nein, warte auf Karte

            ;Setze Karte zurueck
            rcall   rst_tk          ;sende Reset

            ;Setze Adresszaehler auf 24
            ldi     counter,23      ;sende 23 Takte
            rcall   clock_N         ;bis Adresszaehler=24

            ;Lese Hersteller
            rcall   read4           ;lies 4 Bits

            ; ab hier kann erweitert werden

            ;Setze Adresszaehler auf 32
            ldi     counter,4       ;sende 4 Takte
            rcall   clock_N         ;bis Adresszaehler=32

            ;Lese Kartencode
            rcall   read4           ;lies 4 Bits
            ; ab hier kann erweitert werden

            ;Lese Jahr und Monat
            rcall   read8           ;lese 8 Bits
            ; ab hier kann erweitert werden

tk_da:      sbis    PINB,SI         ;Karte eingesteckt?
            rjmp    tk_da           ;warte bis entnommen

            rjmp    main            ;lese neue Karte
```

5.5 Magnetkartenleser

Die äußeren Abmessungen einer Magnetkarte, die in vielen Bereichen zu Identifikationszwecken verwendet wird, und die Lage des Magnetstreifens auf dem Trägermaterial sind genormt. Eine typische Magnetkarte ist in **Bild 5.28** zu sehen. Die äußeren Begrenzungen der Magnetkarte müssen laut ISO 7810 in dem Rahmen liegen, der durch zwei konzentrische Rechtecke mit den Abmessungen 85,72 mm x 54,03 mm und 85,47 mm x 53,92 mm gebildet wird. Die abgerundeten Ecken der Karte weisen einen Radius von 3,18 mm auf.

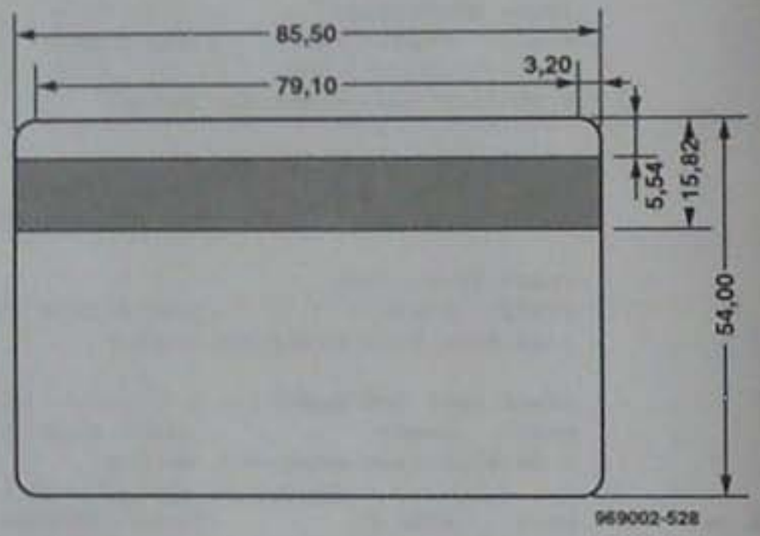


Bild 5.28: Die äußeren Abmessungen der Magnetkarte und die Lage des Magnetstreifens nach ISO 7810.

Der Bereich für den Magnetstreifen wird in Bezug auf die obere und die rechte Kante der Magnetkarte angegeben. Dabei wird angenommen, daß die Karte so orientiert ist, wie in **Bild 5.28** gezeigt. Der Magnetstreifen darf maximal 5,54 mm von der oberen und 2,92 mm von der rechten Bezugskante entfernt liegen. Die Breite des Magnetstreifens hängt davon ab, welche Spuren verwendet werden sollen. Nach den ISO-Vorschriften muß sich der Magnetstreifen bis 11,89 mm unterhalb der oberen Bezugskante erstrecken, wenn man nur Spur 1 und 2 verwendet, und 15,82 mm, wenn man noch zusätzlich die Spur 3 verwendet. Das erste gültige Daten-Bit muß in einem Abstand von 7,44 mm, von der rechten Kartenkante gesehen, auf dem Magnetstreifen stehen. Es ist das erste Bit des Startzeichens. Das letzte Daten-Bit, dabei handelt es sich um das des LRC (Longitudinal Redundancy Check)-Zeichens, muß spätestens 6,93 mm vor der linken Kartenkante stehen.

Innerhalb des Magnetstreifens kann die Information auf drei Spuren geschrieben werden. Diese bezeichnet man als Spur 1, Spur 2 und Spur 3. Die Lage dieser Spuren innerhalb des Magnetstreifens wird in Bezug auf die obere Kante der Karte angegeben. Dabei ist die Lage der Spuren 1 und 2 durch die ISO 7811-4 und die der Spur 3 durch die ISO 7811-5 festgelegt. Die genauen Verhältnisse sind in **Bild 5.29** wiedergegeben.

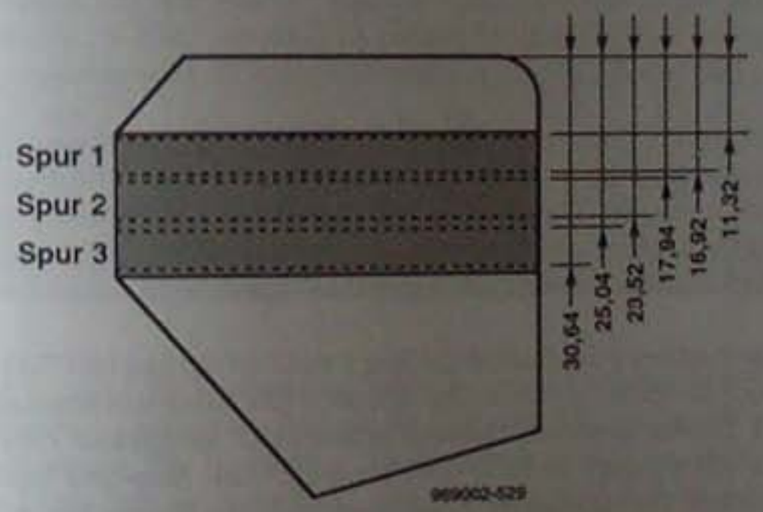


Bild 5.29: Die Lage der drei ISO-Spuren innerhalb des Magnetstreifens.

Die Informationen – bei einer Magnetkarte also die magnetischen Flußwechsel – stehen auf den drei Spuren mit unterschiedlichen Bit-Dichten. Die ISO 7811-4 regelt, daß auf Spur 1 die Bit-Dichte 210 bpi (bpi = bits per inch) und auf Spur 2 75 bpi betragen darf. Auf Spur 3 beträgt die Bit-Dichte 210 bpi, gemäß ISO 7811-5. In den ISO Blättern stehen die angelsächsischen Werte, die nicht dem metrischen Einheitensystem entsprechen. Die Angabe in dieser Form

ist aber in der Magnetkartentechnik üblich. So entsprechen im metrischen Einheitensystem die Angaben 75 bpi 3 Bits/mm und 210 bpi 8,3 Bits/mm.

Die Darstellung der Zeichen auf der Spur 1 wird mit 7-Bit, inklusive einem Paritäts-Bit, welches die Daten-Bits auf ungerade Parität ergänzt, codiert. Damit lassen sich auf dieser Spur alphanumerische Zeichen unterbringen. Auf Spur 2 und 3 werden die Zeichen mit 5-Bit inklusive einem Paritäts-Bit codiert. Durch die 5-Bit breiten Zeichen lassen sich nur numerische Zahlen im BCD-Code darstellen.

Aus der Schreibdichte und der Anzahl der Bits pro Zeichen ergibt sich eine maximale Anzahl von Zeichen, die auf jeder Spur geschrieben werden können. Auf Spur 1 beträgt diese maximal 79, auf Spur 2 maximal 40 und auf Spur 3 maximal 107 Zeichen. Die Angaben gelten inklusive Start-, Stop- und LRC-Zeichen. Das LRC-Zeichen ist die Prüfziffer für den gesamten Datensatz der einzelnen Spuren.

Die Art der Codierung, die man verwendet, hängt von der jeweiligen Anwendung ab. Verwendet wird meist die 5- und 7-Bit-Codierung, die in der ISO 7811-2 beschrieben sind. Bei Verwendung dieser Codierung kann die Anwendung der Magnetkarte, wie bei den Bank-und Kreditkarten, international erfolgen.

Die Codierung der Zeichen auf Spur 2 und 3 erfolgt laut ISO 7811-2 mit 5 Bit inklusive Paritäts-Bit. Mit den 4 Bits lassen sich insgesamt 16 Zeichen darstellen. Darunter befinden sich numerische Ziffern zwischen 0 und 9 im BCD-Code sowie ein Start-, Stop- und Trennzeichen. Der zugehörige Zeichensatz inklusive Paritäts-Bit ist in **Tabelle 5.16** aufgelistet.

Die Codierung der Zeichen auf Spur 1 erfolgt laut ISO 7811-2 mit 7 Bit inklusive Paritäts-Bit. Mit den 6 Bits lassen sich insgesamt 64 Zeichen darstellen. Darunter befinden sich numerische und alphanumerische Ziffern sowie ein Start-, Stop- und Trennzeichen. Der zugehörige Zeichensatz inklusive Paritäts-Bit ist in **Tabelle 5.17** aufgelistet.

Referenzzahl	Paritäts-Bit	b3	b2	b1	b0	Zeichen
0	1	0	0	0	0	0
1	0	0	0	0	1	1
2	0	0	0	1	0	2
3	1	0	0	1	1	3
4	0	0	1	0	0	4
5	1	0	1	0	1	5
6	1	0	1	1	0	6
7	0	0	1	1	1	7
8	0	1	0	0	0	8
9	1	1	0	0	1	9
10	1	1	0	1	0	(a)
11	0	1	0	1	1	:
12	1	1	1	0	0	(a)
13	0	1	1	0	1	=
14	0	1	1	1	0	(a)
15	1	1	1	1	1	?

; : Startzeichen ? : Stopzeichen
= : Trennzeichen (a) : für Hardwaresteuerung benötigt

Tabelle 5.16: Zeichensatz der 5-Bit-Codierung inklusive Paritäts-Bit.

Die 5- und 7-Bit-Codierungen beinhaltet jeweils ein Paritäts-Bit. Mit diesem zusätzlichen Bit kann die Gültigkeit des Zeichens überprüft werden. Die Berechnung des Bits erfolgt so, daß die Anzahl der Daten-Bits, die logisch „1" sind, ungerade wird, d. h die Zeichen werden auf ungerade (odd) Parität ergänzt.

Am Ende jedes Datensatzes einer Spur gibt es ein LRC-Zeichen, welches zur Überprüfung der Gültigkeit des gesamten Datensatzes dient. Die Bitlänge des LRC-Zeichens entspricht der der Zeichen. Die Berechnung der einzelnen Bits geschieht folgendermaßen: Man zählt von allen Zeichen, also inklusive Start- und Stopzeichen, alle Bits, die den Wert „1" an derselben Bitposition haben und setzt dann

Referenz-zahl	Paritäts-Bit	b5	b4	b3	b2	b1	b0	Zeichen
0	1	0	0	0	0	0	0	SP
1	0	0	0	0	0	0	1	(a)
2	0	0	0	0	0	1	0	(a)
3	1	0	0	0	0	1	1	(c)
4	0	0	0	0	1	0	0	$
5	1	0	0	0	1	0	1	%
6	1	0	0	0	1	1	0	(a)
7	0	0	0	0	1	1	1	(a)
8	0	0	0	1	0	0	0	(
9	1	0	0	1	0	0	1	)
10	1	0	0	1	0	1	0	(a)
11	0	0	0	1	0	1	1	(a)
12	1	0	0	1	1	0	0	(a)
13	0	0	0	1	1	0	1	-
14	0	0	0	1	1	1	0	.
15	1	0	0	1	1	1	1	/
16	0	0	1	0	0	0	0	0
17	1	0	1	0	0	0	1	1
18	1	0	1	0	0	1	0	2
19	0	0	1	0	0	1	1	3
20	1	0	1	0	1	0	0	4
21	0	0	1	0	1	0	1	5
22	0	0	1	0	1	1	0	6
23	0	0	1	0	1	1	1	7
24	1	0	1	1	0	0	0	8
25	0	0	1	1	0	0	1	9
26	0	0	1	1	0	1	0	(a)
27	0	0	1	1	0	1	1	(a)
28	0	0	1	1	1	0	0	(a)
29	1	0	1	1	1	0	1	(a)
30	1	0	1	1	1	1	0	(a)
31	0	0	1	1	1	1	1	?
32	0	1	0	0	0	0	0	(a)
33	1	1	0	0	0	0	1	A
34	1	1	0	0	0	1	0	B

Referenz-zahl	Paritäts-Bit	b5	b4	b3	b2	b1	b0	Zeichen
35	0	1	0	0	0	1	1	C
36	1	1	0	0	1	0	0	D
37	0	1	0	0	1	0	1	E
38	0	1	0	0	1	1	0	F
39	1	1	0	0	1	1	1	G
40	1	1	0	1	0	0	0	H
41	0	1	0	1	0	0	1	I
42	0	1	0	1	0	1	0	J
43	1	1	0	1	0	1	1	K
44	0	1	0	1	1	0	0	L
45	1	1	0	1	1	0	1	M
46	1	1	0	1	1	1	0	N
47	0	1	0	1	1	1	1	O
48	1	1	1	0	0	0	0	P
49	0	1	1	0	0	0	1	Q
50	0	1	1	0	0	1	0	R
51	1	1	1	0	0	1	1	S
52	0	1	1	0	1	0	0	T
53	1	1	1	0	1	0	1	U
54	1	1	1	0	1	1	0	V
55	0	1	1	0	1	1	1	W
56	0	1	1	1	0	0	0	X
57	1	1	1	1	0	0	1	Y
58	1	1	1	1	0	1	0	Z
59	0	1	1	1	0	1	1	(b)
60	1	1	1	1	1	0	0	(b)
61	0	1	1	1	1	0	1	(b)
62	0	1	1	1	1	1	0	^
63	1	1	1	1	1	1	1	(a)

(a) : für Hardwaresteuerung benötigt
(b) : für nationale Sonderzeichen reserviert
(c) : reserviert

; : Startzeichen
^ : Trennzeichen
? : Stopzeichen

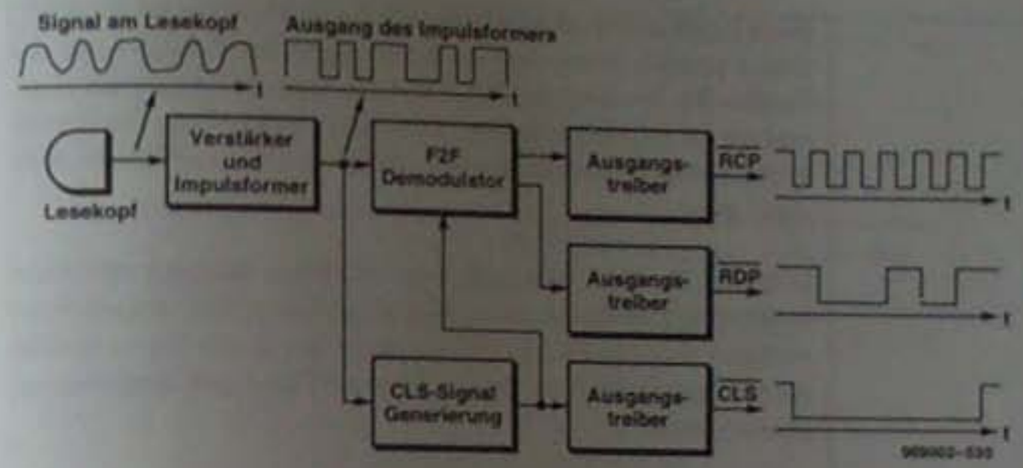
Tabelle 5.17: Zeichensatz der 7-Bit-Codierung inklusive Paritäts-Bit.

das Bit derselben Position des LRC-Zeichens so, daß die Anzahl gerade wird. Dies wiederholt man für alle Bitpositionen (bei der 5-Bit-Codierung b0 bis b3 und bei der 7-Bit-Codierung b0 bis b5), außer der Position des Paritäts-Bits. Das Paritäts-Bit des LRC-Zeichens wird ganz normal wie bei den Datenzeichen auf ungerade Parität ergänzt.

Für das Aufbringen von Informationen auf die Magnetkarte schreibt die ISO-Vorschrift die Wechseltaktschrift vor, die auch unter dem Namen F2F-Modulation bekannt ist. F2F bedeutet „frequency/double frequency", und deutet das Prinzip des Verfahrens an. Diese Modulationsart wurde 1954 von Aiken entwickelt und ermöglicht es, Daten und Takt gemeinsam in den magnetischen Flußwechsel abzulegen. Beim Lesen ist es dann möglich, den Takt wieder zurückzugewinnen. Es handelt sich also dabei um selbstgetaktete Daten. Bei der F2F-Modulation zeigt ein magnetischer Flußwechsel zwischen zwei Taktintervallen eine logische „1" an. Hingegen zeigt das Fehlen eines solchen Flußwechsels eine logische „0" an.

Bevor man die mit dem Magnetkopf vom Magnetband aufgenommenen magnetischen Flußwechsel nutzen kann, müssen sie mit einem F2F-Demodulator Baustein in Daten- und Taktsignale decodiert werden. Das Datensignal wird mit RDP (Read Data Pulse) und das Taktsignal mit RCP (Read Clock Pulse) bezeichnet. Zusätzlich existiert das Signal CLS (Card Load Signal), das ebenfalls aus den magnetischen Flußwechseln gewonnen wird. Es zeigt an, daß eine magnetisierte Karte am Lesekopf entlang gezogen wird, liefert aber keinerlei Information darüber, was auf dem Magnetstreifen steht. Alle Signale werden in der Magnetkartentechnik im allgemeinen in negativer Logik ausgeführt. In **Bild 5.30** ist der schematische Aufbau eines Magnetkartenlesers mit F2F-Decoder und den wichtigsten Signalen dargestellt.

Zum Lesen der Datensignale muß man wissen, wann gültige Daten am Ausgang RDP anliegen. Je nach Typ des Magnetkartenlesers steht ein Signal zur Verfügung, das anzeigt, daß eine Magnetkarte am Lesekopf vorbeigeführt wird. Es wird hier angenommen, daß es sich um das bereits besprochene CLS-Signal handelt. In Frage kommen aber



auch Signale, die bei einigen Geräten durch Mikroschalter oder Photosensoren die Position der Magnetkarte innerhalb des Gerätes anzeigen. Um sicherzustellen, daß gültige Daten am Ausgang anliegen, muß zunächst CLS low sein, ansonsten liegen keine gültige Daten vor. Die Daten RDP müssen dann auf der abfallenden Flanke des Taktes RCP abgetastet werden.

Bild 5.30: Typisches Blockdiagramm eines Magnetkartenlesers.

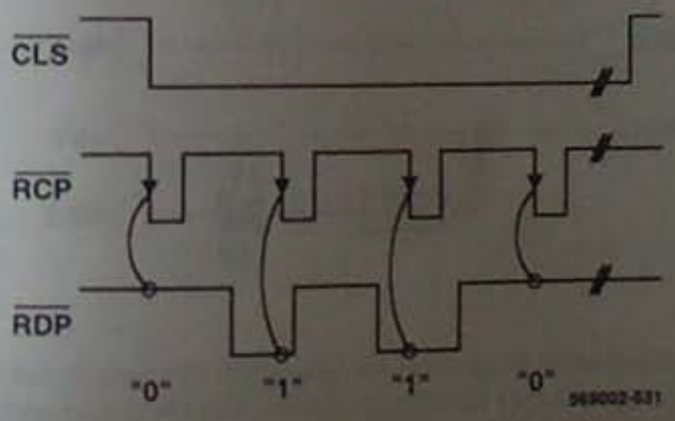


Bild 5.31: Zeitdiagramm der wichtigsten Ausgangssignale CLS, RCP und RDP. Bei negativer Logik liegen auf der abfallenden Flanke des Taktes (RCP) gültige Daten (RDP) an.

Beim Lesen ist darauf zu achten, daß das niederwertigste Bit (LSB) zuerst kommt. Nach dem höchstwertigen Bit (MSB) folgt das Paritäts-Bit. Wechselt der Pegel des CLS-Signals von Low auf High, sind die am Ausgang stehenden Daten nicht mehr gültig. Die Signalverläufe sind nicht zeitkritisch, so daß man diese problemlos abtasten kann.

In **Bild 5.32** ist der Anschluß eines beliebigen Magnetkartenlesers an die AVR-Experimentierplatine gezeigt. Einige Magnetkartenleser verfügen über Open-Collector Ausgänge. Bei diesen Typen muß für die drei Signale RCP, RDP und CLS je ein Pull-Up-Widerstand vorgesehen werden.

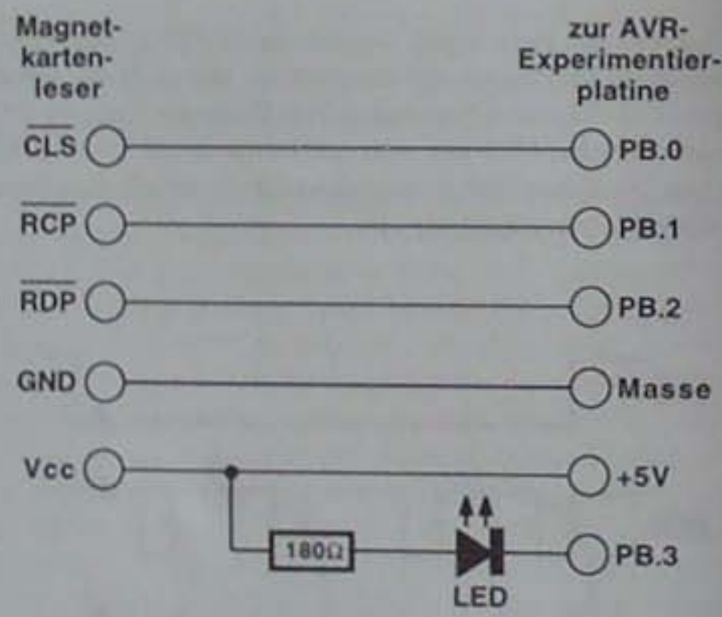


Bild 5.32: Schaltplan zum Anschluß eines Magnetkartenlesers an die AVR-Experimentierplatine.

Das abgedruckte Programm liest den Inhalt der Spur 2, falls eine Codierung nach ISO 7811 Teil 2 vorliegt. Sollte während des Lesen ein Fehler auftreten, leuchtet eine LED auf. Der Leser kann das Programm leicht für eigene Anwendungen modifizieren.


```
;***************************************************************
; Magnetkartenleser
; Version 0.99
;
; File-Name: MKARTE.ASM
;
; Das Programm liest den Inhalt der Spur 2 einer Magnetkarte
; nach ISO 7811 Teil 2. Die Zeichen bestehen aus 4 Bits und
; einem Paritybit.
; Die eingelesenen Zeichen werden im SRAM abgelegt.
;***************************************************************

.device AT90S2313
.include "2313def.inc"


;Konstanten fuer Magnetkarten Interface
.equ        CLS         = PB0       ; PB0 ist CLS
.equ        RCP         = PB1       ; PB1 ist RCP
.equ        RDP         = PB2       ; PB2 ist RDP
.equ        LED         = PB3       ; LED fuer ERROR


;Parameter
.equ        offset      = 0x30      ;Offset fuer ASCII/Werte
.equ        char_len    = 0x04      ;Zeichenlaenge in Bit
.equ        start_char  = 0b00001011 ;Startzeichen
.equ        stop_char   = 0b00001111 ;Stopzeichen

;Variablen
.def        buffer      = r16       ; empfangene Zeichen
.def        counter     = r17       ; Zaehler
.def        parity      = r18       ; Paritybit
.def        temp        = r19

.CSEG
.ORG        0x00                    ; Programm beginnt bei 0
            rjmp    main            ; Starte Hauptprogramm


;***************************************************************
;Subroutine cls_start
;Testet das CLS-Signal. Die Routine wird verlassen, falls
;CLS low ist.
;***************************************************************
```

```
cls_start:  sbic    PINB,CLS        ;Ist CLS low?
            rjmp    cls_start       ;nein, warte
            ret                     ;ja, springe zurueck

;***************************************************************
;Subroutine cls_low
;Testet, ob das CLS-Signal immer noch low ist. Ist CLS waehrend
;die Magnetkarte durchgezogen wird high, ist ein Fehler
;aufgetreten.
;***************************************************************

cls_low:    sbic    PINB,CLS        ;Ist CLS low?
            rjmp    err_LED         ;nein, Fehler!
            ret                     ;ja, springe zurueck

;***************************************************************
;Subroutine read_bit
;Diese Routine liest ein Bit vom Magnetkartenleser ein.
;Es ist die negative Logik zu beachten.
;Das gelesene Bit wird im Carryflag (richtig) uebergeben.
;***************************************************************

read_bit:   clc                     ;bereite Carry vor
RCP_high:   sbic    PINB,RCP        ;warte auf Flanke
            rjmp    RCP_high        ;
            sbis    PINB,RDP        ;werte RCP aus
            sec                     ;low empfangen (negiert!)
RCP_low:    sbis    PINB,RCP        ;warte, bis RCP wieder
            rjmp    RCP_low         ;high
            ret

;***************************************************************
;Subroutine start
;Diese Routine wartet auf das Startzeichen.
;Die Routine wird verlassen, wenn das Startzeichen empfangen
;wurde.
;***************************************************************

start:      rcall   cls_low         ;teste CLS
            rcall   read_bit        ;lies ein Bit ein
            ror     buffer          ;Bit retten
            mov     temp,buffer     ;
            swap    temp            ;
            andi    temp,0b00001111 ;unteres Nibble maskieren
            cpi     temp,start_char ;Start empfangen?
```



```
            brne    start           ;nein, empfange weiter
            rcall   read_bit        ;lies Paritybit
            brcs    err_LED         ;Fehler, fall Carry=1
            ret


;***************************************************************
;Subroutine read_char
;Diese Routine liest ein Zeichen vom Magnetkartenleser ein.
;Das gelesene Zeichen wird im Register buffer gespeichert und
;steht zur weiteren Bearbeitung bereit. Es wird immer auf die
;richtige Paritaet geprueft.
;Es wird solange gelesen, bis das Stopzeichen empfangen wird.
;Die Routine wird verlassen, wenn das Stopzeichen empfangen
;wird.
;***************************************************************

read_char:  clr     parity          ;bereite Parity vor
            ldi     counter,char_len;bereite Zaehler vor
next_bit:   rcall   cls_low         ;teste CLS
            rcall   read_bit        ;lies ein Bit ein
            brcc    no_inc          ;zaehle 1-Bits
            inc     parity          ;Parity erhoehen
no_inc:     ror     buffer          ;Bit retten
            dec     counter         ;4 Bits gelesen?
            brne    next_bit        ;nein, lies weiter
            rcall   read_bit        ;lies Paritybit
            brcc    no_inc2         ;teste gelesenes und
            inc     parity          ;berechnetes Parity
no_inc2:    sbrs    parity,0        ;Parity ok?
            rjmp    err_LED         ;nein, Fehler!
            swap    buffer          ;werte Zeichen aus
            andi    buffer,0b00001111;unteres Nibble maskieren

            ;Ab hier steht das gelesene Zeichen
            ;richtig und auf gueltige Paritaet gespruefft
            ;im Register buffer und werden im SRAM abgelegt

            st      Z+,buffer       ;Zeichen in SRAM ablegen
                                    ;und Pointer erhoehen


            cpi     buffer,stop_char;Stop empfangen?
            brne    read_char       ;ja, beende Routine
                                    ;nein, empfange weiter
            ret
```

```
;***************************************************************
;Subroutine err_LED
;Diese Routine schaltet die LED ein, falls ein Fehler
;auftritt.
;***************************************************************

err_LED:    cbi     PORTB,LED       ;LED einschalten
            rjmp    err_LED         ;Endlosschleife
            ret


;***************************************************************
;Hauptprogramm
;***************************************************************
main:       ldi     temp,RAMEND     ; setze Stack-Pointer
            out     SPL,temp        ; an das SRAM-Ende

            ldi     temp,0b11111111 ;PORTB initialisieren
            out     PORTB,temp
            ldi     temp,0b00000000 ;Datenrichtung initialisieren
            out     DDRB,temp

            ldi     ZL,$60          ;Daten-Pointer initialisieren
            clr     ZH

            sbi     PORTB,LED       ;LED ausschalten
            clr     buffer          ;buffer vorbereiten

            rcall   cls_start       ;Karte da?
            rcall   start           ;Startzeichen abwarten
            rcall   read_char       ;liest Zeichen ein, bis
                                    ;Stopzeichen empfangen

endlos:     rjmp    endlos          ;Endlosschleife
```


5.6 Serielle Kommunikation

In vielen Mikrocontrollerapplikationen ist es notwendig, Daten z. B. mit einem PC auszutauschen. Dazu verwendet man eine serielle Schnittstelle. Bis auf den AT90S1200 besitzen alle AVR-Mikrocontroller zu diesem Zweck einen Universal Asynchronous Receiver Transmitter (UART). Die Datenübertragung erfolgt seriell und asynchron, d. h. es wird neben den Daten kein Takt auf einer zusätzlichen Leitung übertragen.

Die Übertragung eines Bytes erfolgt überlicherweise in 10 Bits: Ein Startbit, acht Datenbits und ein Stopbit. Dabei wird das LSB (Least Significant Bit) des Datenbytes zuerst übertragen. Start- und Stopbit dienen dazu, im Empfänger den hereinkommenden Datenstrom besser synchronisieren zu können. Dabei besteht das Startzeichen aus einer High-Low-Flanke auf der Datenleitung. Daran anschließend muß die Datenleitung eine bestimmze Zeit, die sich aus der Datenübertragungsrate ergibt, auf Low bleiben. Das Stopzeichen besteht aus einer Low-High-Flanke und anschließend muß die Datenleitung wieder für eine feste Zeit (s. o.) auf High bleiben. Bei einer Datenübertragungsrate von 9600 Baud (oder auch bits per second, bps) ist die zeitliche Ausdehnung eines einzelnen Bits genau 1/9600 s also präzise 104 µs. Zur Übertragung eines Bytes inklusive der Start- und Stopbits benötigt man demnach 10 x 104 µs = 1,04 ms (siehe **Bild 5.33**).

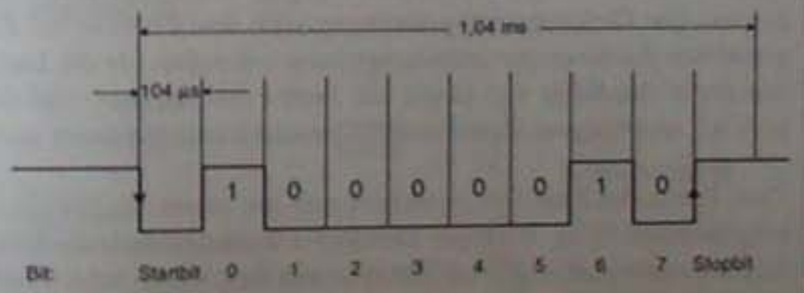


Bild 5.33: *Serielle asynchrone Datenübertragung. Dargestellt sind: Startbit, acht Datenbits (01000001 Hex 'A') und ein Stopbit bei 9600 Baud.*

Für die Signalpegel bei der seriellen Datenübertragung existiert eine Norm mit dem Namen RS-232. Bei der Datenübertragung nach RS-232 wird einer logischen „1" ein Signalpegel von −5 V bis −15 V, und einer logischen „0" ein Signalpegel von +5V bis +15V zugeordnet. Da Mikrocontroller üblicherweise nur TTL-Pegel zulassen, muß eine Pegelwandlung vorgenommen werden. Dazu verwendet man z. B. einen MAX232 der Firma MAXIM o. ä., der nur mit 5 V versorgt wird und über eine Ladungspumpe die Pegel nach RS-232 generiert. Auf der AVR-Experimentierplatine ist bereits ein solcher Pegelwandler vorgesehen, der über den Schalter S1 an die Ports PD0 und PD1 geschaltet werden kann. Die Sub-D-Buchse X1 sorgt für den Anschluß an die Außenwelt.

Das folgende Programm zeigt, wie man bei einem AT90S2313 eine serielle asynchrone Schnittstelle initialisiert und mit dieser, Daten zwischen Mikrocontroller und PC austauschen kann. Zur Demonstration wird das über die serielle Schnittstelle empfangene Zeichen gleich wieder zurückgesendet. Ferner wird der empfangene Wert des Bytes am PWM-Kanal ausgegeben. Dabei ist den Werten 00_{hex} bis FF_{hex} eine Gleichspannung zwischen 0 V und 5 V zugeordnet. Damit am PB3-Pin aus dem PWM-Signal eine Gleichspannung abgegriffen werden kann, ist ein einfaches RC-Filter nachgeschaltet. In **Bild 5.34** sind die Signale RX, TX, PB3 und Filter dargestellt. Man sieht an PB3 das PWM-Signal, bevor ein neuer Wert seriell über RX empfangen wurde. Das Signal nach dem RC-Filter ist in der Kurve „Filter" dargestellt. Hier sieht man die zum PWM-Signal zugehörige Gleichspannung. Nachdem ein neuer Wert über RX empfangen wurde, ändert sich das PWM-Signal (ca. in der Mitte des Bildes). Die Gleichspannungsänderung nach dem Filter ist bei der gewählten Zeitbasis nur andeutungsweise erkennbar, da die Laufzeit durch das Filter viel länger ist. Ferner erkennt man, daß das über RX empfangene Signal über TX wieder zurückgesendet wird.

Zum Testen der Routinen verwendet man am besten ein Terminalprogramm am PC (z. B. Hyper-Terminal o. ä.) und schließt die AVR-Experimentierplatine über ein Kabel an eine freie COM-Schnittstelle des PCs an. Tippt man nun ein Zeichen auf der Tastatur des PCs, so wird das vom AVR empfangene Zeichen wieder an den PC zurück-

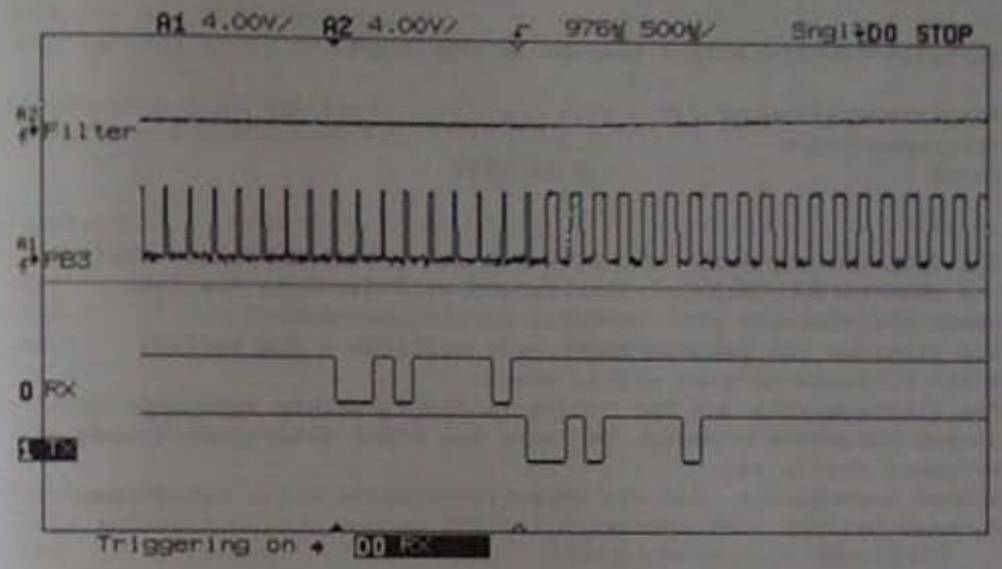


Bild 5.34: PWM-Signal vor (Kurve PB3) und nach Filterung (Kurve Filter) sowie seriell empfangenes (Kurve RX) und gesendetes Byte (Kurve TX)

gesendet und am Bildschirm angezeigt. Hat man die ECHO-Funktion des Terminalprogramms aktiviert, wird am Bildschirm das Zeichen zweimal angezeigt: Das erste Zeichen rührt vom lokalen Echo im PC her, das zweite Zeichen ist das tatsächlich gesendete und wieder empfangene.

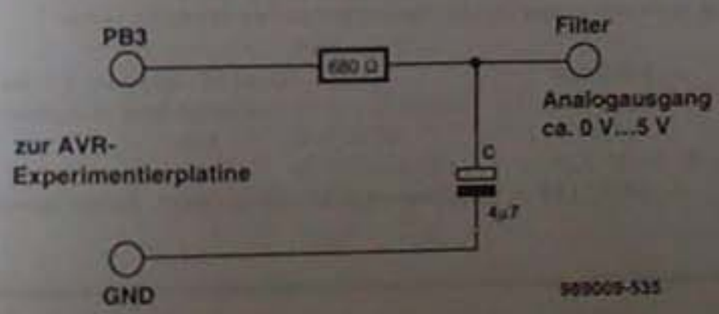


Bild 5.35: Schaltplan zur Darstellung der empfangenen Werte über den PWM-Kanal (die Bauteile für die serielle Schnittstelle sind bereits auf der AVR Experimentierplatine integriert).

```
;***************************************************************
;
;Serielle-Uebertragung / Ausgabe am PWM-Kanal
;
;Programmbezeichnung        :VP1
;Programmversion            :1.0
;Datum                      :15.09.1998
;
;
;Das Programm empfaengt ein Byte von der seriellen Schnittstelle
;und gibt es am PWM-Kanal aus. Danach wird das Byte als Echo
;ueber die serielle Schnittstelle zurueckgesendet.
;Das Programm ist bewusst etwas mehr gegliedert als es fuer
;diese einfache Aufgabe noetig waere.
;Die Flagsteuerung und das Senden im Hauptprogramm sind bei
;dieser Aufgabe ein Umweg, der aber bei etwas anspruchsvolleren
;Aufgaben noetig ist.
;Andere Anwendungen, die ein Uebertragungsprotokolls benoetigen,
;koennen bei der vorliegenden Gliedrung bequem ergaenzt werden.
;
;***************************************************************

.device    AT90S2313
.include "2313def.inc"


;***************************************************************
;Baudrate berechnen:
;Die Baudrate berechnet sich nach der Formel: (Siehe Datenblatt)
;Baudrate = Quarzfrequenz / 16*(UBRR-1)
;Da UBRR nur ganze Zahlen darstellen kann, koennen bei einer
;gegebenen Quarzfrequenz nicht alle moeglichen Baudraten
;realisiert werden. Die angegebene Formel zur Berechnung der
;Baudrate gilt nur dann, wenn das Ergebnis sehr nahe bei einer
;ganzen Zahl liegt. Vor der Berechnung sollte kontrolliert
;werden, ob die gewuenschte Baudrate moeglich ist.
;***************************************************************

.equ    fck          = 4000000                  ;Quarzfrequenz in Hz
.equ    baudrate     = 9600                     ;gewuenschte Baudrate


.equ    baudkonst    = (fck/(16 * Baudrate))-1 ;UBRR-Wert berechnen
```



```
;Konstanten fuer RS232-Interface
;PD0 ist der RS232 Empfangspin
;PD1 ist der RS232 Sendepin

;Konstanten fuer PWM-Kanal
;PB3 ist der PWM-Ausgabepin


;Variablen
.def        w       = r19          ;Erstes Arbeitsregister
.def        wi      = r20          ;Interrupt Arbeitsregister
.def        rs_buf  = r21          ;Buffer fuer ein RS232 Zeichen
.def        a_flag  = r25          ;Flagregister

;Flagbelegung bei a_flag
.equ        rs_recv   = 0          ;1 -> Ein Byte wurde empfangen

;Vektortabelle
.CSEG
.ORG        $0000                  ;Programm beginnt bei $0000
            rjmp    main           ;Programmstart bei main
.ORG        $0007                  ;RS232 Empfangsinterrupt
            rjmp    rs232_recv     ;RS232 Behandlungsroutine


;***************************************************************
;Subroutine hdw_init
;Die I/O-Port, der UART und der PWM-Kanal werden initialisiert
;In DDRx   0 -> Input   1 -> Output
;Die serielle Schnittstelle wird auf 8N1 mit der vorgewaehlten
;Baudrate initialisiert.
;Der PWM Kanal wird auf 8 Bit PWM initialisiert. (Nur bei Bedarf!)
;***************************************************************

hdw_init:   ldi     w,0b11111111   ;Initialisiere Port D
            out     PORTD,w        ;
            ldi     w,0b11111110   ;Init Port D Datenrichtung
            out     DDRD,w         ;

            ldi     w,0b11111111   ;Initialisiere Port B
            out     PORTB,w        ;
            ldi     w,0b11111111   ;Init Port B Datenrichtung
            out     DDRB,w         ;
```

```
;Ab hier wird das UART initialisiert
uart_init:  ldi     w,baudkonst     ;Baudrate laden
            out     UBRR,w          ;
            ldi     w,0b00011000    ;8 Bits Senden/Empfangen
            out     UCR,w           ;
            sbi     UCR,RXCIE       ;Empfangsinterrupt freigeben

;Ab hier: PWM Ausg. initialiseren
pwm_init:   ldi     w,0b10000001    ;8 Bit PWM waehlen
                                    ;PWM-PIN ist OC1
            out     TCCR1A,w        ;
            ldi     w,0             ;PWM-Anfangswert initialisieren
            out     OCR1AH,w        ;
            ldi     w,1             ;
            out     OCR1AL,w        ;
            ldi     w,0b00000001    ;T1 Eingangsteiler = 1 und T1 start
            out     TCCR1B,w        ;

            sei                     ;Alle Ints freigeben
            ret



;*****************************************************************
;Subroutine rs_recv_up
;Hier erfolgt die Behandlung des empfangenen Bytes. Die Ausgabe
;erfolgt am PWM Kanal. Bei der PWM-Ausgabe muss die Ausgabe
;zum Port deaktiviert, und die PWM-Ausgabe aktiviert werden.
;Bei der Hardwareinitialisierung hdw_init sind weitere Anweisungen
;zu Aktivieren! Ferner wird das empfangene Byte ueber ein Echo
;seriell zurueckgesendet.
;*****************************************************************

rs_recv_up: cbr     a_flag,1<<rs_recv;Empfangsflag zuruecksetzen
            mov     w,rs_buf        ;dann das Byte
            out     PORTB,w         ;zum Port ausgeben (Kein PWM)
            out     OCR1AL,w        ;zum PWM Kanal augeben
            rcall   rs_send         ;und seriell zuruecksenden
                    ret

;*****************************************************************
;Subroutine rs_send
;Das zu sendende Byte wird im Register w uebergeben
;*****************************************************************
```



```
rs_send:    sbis    USR,UDRE        ;Pruefe, ob der Sender frei
            rjmp    rs_send         ;falls ja dann
            out     UDR,w           ;ausgeben
            ret


;*****************************************************************
;Interrupt-Routine rs232_recv
;Diese Interrupt-Routine wird aufgerufen, falls ein Byte am
;UART empfangen wurde.
;*****************************************************************

rs232_recv: in      wi,SREG         ;CPU-Status sichern
            push    wi

            in      wi,UDR          ;Byte vom Empfänger laden
            mov     rs_buf,wi       ;und zwischenspeichern
                                    ;dann
            sbr     a_flag,1<<rs_recv;Empfangflag setzen
            pop     wi              ;CPU-Status restaurieren
            out     SREG,wi
            reti                    ;Beende die Interruptbehandlung


;*****************************************************************
;Hauptprogramm
;Prueft, ob ein Byte am UART empfangen wurde. Falls ja, wird es
;weiter behandelt. Der UART-Empfang ist interruptgesteuert.
;*****************************************************************

main:       ldi     w,RAMEND        ;Stack initialisieren
            out     SPL,w
            clr     a_flag          ;Flag zuruecksetzen
            rcall   hdw_init        ;Initialisiere die Hardware

endlos:     sbrc    a_flag,rs_recv  ;Pruefe ob Byte empfangen wurde
            rcall   rs_recv_up      ;falls ja, behandeln
            rjmp    endlos          ;Endlosschleife
```

5.7 Cyclic Redundancy Check (CRC)

Eine effektive Möglichkeit der Fehlererkennung bei seriellen Übertragungen o. ä. bietet die Berechnung einer CRC (Cyclic Redundancy Check)-Prüfsumme. Die Idee beim CRC besteht darin, die Daten (alle Bytes oder Worte eines Datensatzes, nicht ein einzelnes Byte oder Wort) als eine sehr große Zahl aufzufassen, die durch einen bestimmten Wert dividiert wird. Nun ist es nicht einfach vorherzusagen, aus wie vielen Bits der Quotient, also das Ergebnis, besteht. Die Länge der CRC-Prüfsumme würde von den Daten und dem Divisor abhängen, und man könnte somit keine genaue Aussage treffen.

Statt des Quotienten kann man aber genausogut den Divisionsrest als Prüfsumme verwenden, da dies ebenso willkürlich ist wie die Verwendung des Quotienten. Der Divisionsrest hat einen entscheidenden Vorteil: Die Bitlänge des Divisors bestimmt die Bitlänge des Divisonsrests. Man kann mathematisch zeigen, daß der Divisionsrest genau ein Bit kürzer ist als der Divisor. Ferner kann man zeigen, daß der Divisor mit einer „1" beginnen und enden muß. Da man bei einer Division den Divisor vom Dividenden subtrahiert (man bedenke die Modulo-2-Arithmetik!), muß das höchstwertige Bit eine „1" sein, damit die erste Stelle des Rests Null wird. Im Zusammenhang mit CRC wird statt vom Divisor sehr häufig vom Generatorpolynom gesprochen. Prinzipiell ist das nur ein anderes Wort für dieselbe Sache.

Im weiteren beschränken wir uns auf eine CRC-8 Prüfsumme, d. h. der Rest oder die Prüfsumme ist acht Bit lang. Das Generatorpolynom lautet dann:

$$x^8 + x^5 + x^4 + 1$$

Dieser Schreibweise entnimmt man, daß sich nur an den Stellen 2^8, 2^5, 2^4 und 2^0 Einsen befinden (= 100110001 binär). Das ist eine

neun Bit lange Zahl. Als Divisor wird aber aus praktischen Erwägungen nur ein acht Bit langes Polynom ($= 00110001_b = 31_{hex}$) verwendet. Wie ist dieser Divisor, der mit einer „0" beginnt, mit der Forderung in Einklang zu bringen, daß der Divisor mit einer „1" beginnen muß? Wie schon angedeutet zwingt eine führende „1" bei der Subtraktion das höchstwertige Bit des Divisionsrests zu „0". Das ist immer der Fall, so daß dies in der Realisierung eines CRC-Prüfsummengenerators berücksichtigt wird und die führende „1" im Divisor ignoriert werden kann.

Die Realisierung eines CRC-Prüfsummengenerators ist sehr einfach, denn die Berechnung erfodert nur die Divion, was sich softwaremäßig durch zyklisches Schieben erreichen läßt, und die Subtraktion, was sich durch die Bildung einer Exklusiv-Oder-Verknüpfung (XOR) ausführen läßt. Es sei daran erinnert, daß die Subtraktion in der Modulo-2-Arithmetik einer Exklusiv-Oder-Verknüpfung entspricht. In **Bild 5.36** ist eine Realisierung eines CRC-Prüfsummengenerators mit Schieberegistern und OR-Gattern zu sehen. Die eingekreisten Pluszeichen stellen die XOR-Gatter dar. Jedes Rechteck ist ein Flipflop des Schieberegisters.

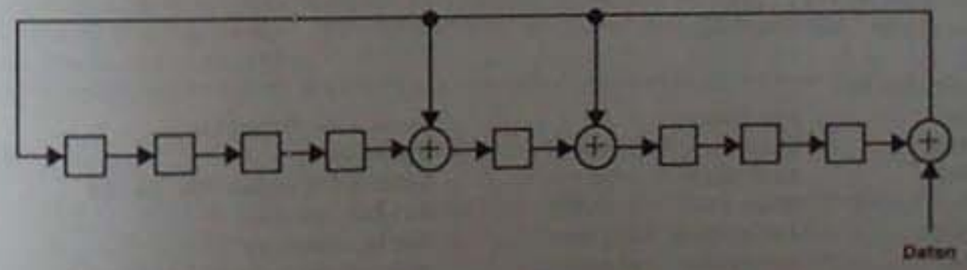


Bild 5.36: Realisierung eines CRC-8 Prüfsummengenerators.

Bevor mann anfangen kann, eine CRC-Prüfsumme zu berechnen, muß man den CRC-Akkumulator (die Flipflops des Schieberegisters) noch mit einem Wert vorladen. Üblicherweise verwendet man

für alle Stellen den Wert null. Man beachte, daß von dieser Vorbelegung letztendlich das Ergebnis abhängt!

Durch die Überprüfung mittels CRC-8 lassen sich u. a. folgende Fehler erkennen:

- eine ungerade Anzahl von Bitfehlern innerhalb des gesamten Datensatzes,
- alle doppelten Bitfehler innerhalb des gesamten Datensatzes und
- Fehler, die innerhalb eines 8-Bit-Fensters liegen.

```
;*****************************************************
; Cyclic-Redundancy-Check
;
; File-Name: crc.ASM
;
; Das Programm berechnet eine CRC-8 Pruefsumme. Als Beispiel
; werden sieben Datenbytes aus einer Tabelle genommen und die
; zugehoerige CRC-8 Pruefsumme gebildet.
;*****************************************************

.device AT90S2313
.include "2313def.inc"

;Variablen
.def        zeichen    = r0        ; Byte aus Tabelle
.def        buffer     = r16       ; Datenbyte
.def        bit_cntr   = r17       ; Datenbitzeiger in FR 10
.def        hilf       = r18       ; Hilfsregister
.def        data_cntr  = r19       ; Daten-Zaehler
.def        crc_akku   = r20       ; Akku des CRC-Generators
.def        poly       = r21       ; CRC-8 Polynom (da kein EORI!)
.def        temp       = r22

.CSEG
.ORG        0x000                  ; Programm beginnt bei 0
            rjmp    main           ; Starte Hauptprogramm
```



```
;*****************************************************
;Tabelle mit den Datenbytes zu denen der CRC-8 berechnet wird
;*****************************************************

Tabelle:    .db 0x02,0x1C,0xB8,0x01,0x00,0x00,0x00

;*****************************************************
;Subroutine crc_8
;Diese Subroutine berechnet den CRC-8 eines Bytes, das
;bitweise in den CRC-Akku geschoben wird.
;*****************************************************

crc_8:      clr     hilf            ;Register „hilf" vorbereiten
            clc                     ;Carry loeschen
            ror     crc_akku        ;CRC-Akku schieben
            brcc    MSB_low         ;MSB = 1?
            inc     hilf            ;ja, setze Flag
MSB_low:    sbrc    buffer,7        ;Bit = 1?
            inc     hilf            ;ja
            sbrs    hilf,0          ;XOR=1?
            rjmp    no_add
            eor     crc_akku,poly
no_add:     ret

;*****************************************************
;Hauptprogramm
;Es werden sieben Bytes aus der Tabelle gelesen und bitweise
;in den CRC-Akku geschoben. Die CRC-8 Pruefsumme der
;Datenbytes betraegt „A2" und steht am Ende der Berechnung
;im Register crc_akku.
;*****************************************************

main:       ldi     temp,RAMEND     ;setze Stack-Pointer
            out     SPL,temp        ;an das SRAM-Ende
            ldi     poly,0b10001100 ;CRC-8 Polynom (rueckwaerts)
            clr     crc_akku        ;Register vorbereiten
            clr     buffer          ;Register vorbereiten

            ldi     ZL,LOW(Tabelle*2);Zeiger auf Tabellen-
            ldi     ZH,HIGH(Tabelle*2);anfang setzen

            ldi     data_cntr,7     ;7 Bytes aus Tabelle lesen
```

```
next_data:  lpm                 ;Hole Byte aus Tabelle
            ldi     bit_cntr,8  ;Bitweise schieben
loop:       ror     zeichen
            ror     buffer
            rcall   crc_8       ;neuen CRC-8 berechnen
            dec     bit_cntr
            brne    loop        ;alle Bits geschoben?
            clc
            inc     ZL          ;Low-Zeiger um 1 erhoehen
            brcc    no_carry    ;kein Uebertrag von ZL
            inc     ZH          ;ZH erhoehen, da Uebertrag von ZL
no_carry:   dec     data_cntr   ;alle 7 Bytes berechnet?
            brne    next_data
endlos:     rjmp    endlos      ;CRC-8 steht in crc_akku
```


Anhang

A.1 Inhalt der CD

Auf der beiliegenden CD befinden sich folgende Dateien:

Im Verzeichnis \App (Applikationsschriften)

\App	doc0931.pdf	:AVR000	Register and Bit-Name Definitions for the AVR Microcontrollers
	doc0932.pdf	:AVR100	Accessing the AT90S1200 EEPROM
	doc0933.pdf	:AVR102	Block Copy Routines
	doc0934.pdf	:AVR128	Setup and Use the Analog Comparator
	doc0935.pdf	:AVR190	Power Up Considerations
	doc0936.pdf	:AVR200	Multiply and Divide Routines
	doc0937.pdf	:AVR202	16-Bit Arithmetics
	doc0938.pdf	:AVR204	BCD Arithmetics
	doc0939.pdf	:AVR220	Bubble Sort
	doc0940.pdf	:AVR222	8-Point Moving Average Filter
	doc0941.pdf	:AVR304	Half Duplex Interrupt Driven Software UART
	doc0942.pdf	:AVR400	Low Cost A/D Converter
	doc0943.pdf	:AVR400	Low Cost A/D Converter
	doc0951.pdf	:AVR302	Software I2C Slave Implementation
	doc0952.pdf	:AVR305	Half Duplex Compact Software UART

doc0953.pdf	:AVR401	8-Bit Precision A/D Converter
doc0954.pdf	:AVR300	Software I²C Master Interface
doc1022.pdf	:AVR	Assembler User Guide
doc1108.pdf	:AVR320	Software SPI Master
doc1143.pdf	:AVR236	CRC Check of Program Memory
doc1181.pdf	:AVR360	Step Motor Controller
doc1232.pdf	:AVR240	4 x 4 Keypad - Wake up on Keypress
doc1235.pdf	:AVR313	Interfacing the PC AT Keyboard

Im Verzeichnis \AVRAsm befinden sich die Include-Dateien für die AVR-Mikrocontroller. Es ist darauf zu achten, daß der richtige Pfad im Assembler-Source-File angegeben wird.

\AVRAsm	1200def.inc
	2313def.inc
	2323def.inc
	2333def.inc
	2343def.inc
	4414def.inc
	4433def.inc
	4434def.inc
	8515def.inc
	8535def.inc
	M103def.inc
	M603def.inc

Im Verzeichnis \Bin befinden sich die Softwaretools

\Bin	aprogdos.exe	DOS-Programm zum AVR-Programmer
	aprogwin.exe	Windows95/NT-Programm zum AVR-Programmer
	archive.exe	Datenkomprimierprogramm
	astudio.exe	AVR-Studio (AVR-Simulator und Emulator-Tool)
	avr.exe	AVR-Assembler und AVR-Simulator (der Simulator wird nicht mehr unterstützt!)

Im Verzeichnis \Data befinden sich Datenblätter der AVR-Mikrocontroller

\Data	0838.pdf	AT90S1200
	0839.pdf	AT90S2313
	1042.pdf	AT90S2333/AT90LS2333/AT90S4433/AT90LS4433
	2323_f.pdf	AT90S2323 Rev F and AT90LS2323 Rev F ERRATA SHEET
	doc0840.pdf	AT90S4414 Preliminary
	doc0841.pdf	AT90S8515 Preliminary
	doc0855.pdf	AVR Enhanced RISC Microcontrollers
	doc0945.pdf	ATmega603/ATmega603L/ATmega103/ATmega103L
	doc1004.pdf	AT90S2323/AT90LS2323/AT90S2343/AT90LS2343 Preliminary
	doc1015.pdf	Understanding the AVR ICEPRO I/O Registers
	doc1019.pdf	AVR Studio User Guide
	doc1024.pdf	AVR AT90ICEPRO User Guide
	doc1041.pdf	AT90S4434/AT90LS4434/AT90S8535/AT90LS8535 Advance Information
	doc1190.pdf	AT90S1200/A Rev. F Errata Sheet
	doc1192.pdf	AT90S/LS2323 Rev. F Errata Sheet
	doc1193.pdf	AT90S/LS2343 Rev. F Errata Sheet
	doc1196.pdf	AT90S/LS8535 Rev. D Errata Sheet
	doc1197.pdf	ATmega103L Rev. F/G, ATmega103 Rev. G Errata Sheet
	instruction.pdf	Instruction Set

Auf der CD-ROM befindet sich auch ein Ordner \Acroread mit dem „Acrobat Reader", der zum Lesen der PDS-Dateien erforderlich ist.

Im Verzeichnis \Projekte befinden sich die Assembler-Sourcen zur Programmsammlung in Kapitel 5.

\Projekte	touchmem.asm	: Abschnitt 5.1
	i2cbus.asm	: Abschnitt 5.2
	lcd.asm	: Abschnitt 5.3
	tkl.asm	: Abschnitt 5.4
	mkarte.asm	: Abschnitt 5.5
	seriell.asm	: Abschnitt 5.6
	crc.asm	: Abschnitt 5.7

Im Verzeichnis \Source befinden sich die Assembler-Sourcen zu den Applikationsschriften (gleiche Nummerierung).

\Source	avr100.asm
	avr102.asm
	avr128.asm
	avr200.asm
	avr200b.asm
	avr202.asm
	avr204.asm
	avr220.asm
	avr222.asm
	avr235.asm
	avr300.asm
	avr302.asm
	avr304.asm
	avr305.asm
	avr320.asm
	avr400.asm
	avr401.asm
	avr910.asm

A.2 Bezugsquellen

Atmel GmbH

ATMEL ist der Hersteller der AVR-Mikrocontroller und von seriellen EEPROMs. Ferner liefert die Firma ATMEL den AT90ICEPRO Emulator. Von der Home Page kann man sich die aktuellsten Versionen der AVR-Softwaretools downloaden. Ein ATMEL Distributor ist z. B. Ineltek.

ATMEL GmbH
Kufsteiner Str. 35
83064 Raubling
Tel.: 0 80 35/9 01 80
Fax: 0 80 35/90 18 33
http://www.atmel.com

Dallas Semiconductor

Dallas Semiconductor ist der Hersteller der Touch-Memories. Ein Dallas Semiconductor Distributor ist z. B. Future Electronics.

Dallas Semiconductor
Am Söldnermoos 17
85399 Halbergmoos
Tel.: 08 11/60 09 60
Fax: 0811/6 00 96 20
http://www.dalsemi.com

Elektronik Laden Detmold

Der Elektronik Laden Detmold vertreibt den in Kapitel 4 beschriebenen AVR-Programmer. Ferner ist dort auch die AVR-Experimentierplatine aus Kapitel 5 inklusive einem AT90S2313 zu haben.

Elektronik Laden Detmold
W.-Mellies-Straße 88
32758 Detmold
Tel.: 0 52 32/81 71
Fax: 0 52 32/8 61 97
http://www.elektronikladen.de

Equinox Technologies

Die Firma Equinox Technologies in England liefert eigene Programmiergeräte und Experimentierboards zu den AVR-Mikrocontrollern. Ferner wird ein AVR-BASIC angeboten.

Equinox Technologies UK Limited
3 Atlas House
St. Georges Square
Bolton BL1 2HB
England
Tel.: 0044-1204-529000
Fax: 0044-1204-535555
http://www.equinox-tech.com

Future Electronics Deutschland GmbH

Die Firma Future Electronics ist u. a. Distributur für Dallas Semiconductor und liefert die Touch-Memory Bausteine.

Future Electronics Deutschland GmbH
Münchner Straße 18
85774 Unterföhring
Tel.: 0 89/95 72 70
Fax: 0 89/95 72 71 40

Hewlett-Packard GmbH

Über HP DIRECT kann das Mixed-Signal-Oszilloskop HP 54645D, das in Kapitel 4 besprochen wurde, bezogen werden.

Hewlett-Packard GmbH
Schickardstraße 2
71034 Böblingen
Tel.: 0 70 31/14 63 33
Fax: 0 70 31/14 63 36
http://www.hp.com

IAR Systems GmbH

Die Firma IAR Systems bietet einen eigenen Assembler und C-Compiler für die AVR-Mikrocontroller an. Ferner gehört eine Arbeitsumgebung für die verschiedenen Softwaretools zum Lieferumfang.

IAR Systems GmbH
Brucknerstraße 27
81677 München
Tel.: 0 89/4 70 60 22
Fax: 0 89/4 70 95 65
http://www.iar.se

INELTEK GmbH

Die Firma INTELTEK ist ein Distributor für ATMEL Produkte. Hier können alle AVR-Mikrocontroller sowie alle ATMEL Soft- und Hardwaretools bestellt werden.

INETELK GmbH
Hauptstraße 45
89522 Heidenheim
Tel.: 0 73 21/9 38 50
Fax: 0 73 21/93 85 95
http://www.ineltek.com

Ingenierbüro Yahya

Das Ingenieurbüro Yahya ist im Großraum Düsseldorf im Bereich MSR tätig. Für die AVR-Mikrocontroller wird ein eigenes Programmiergerät angeboten.

Ingenieurbüro Yahya
Robert-Schumann-Straße 2A
41812 Erkelenz
Tel.: 0 24 31/64 44
Fax: 0 24 31/45 95

A.3 Literaturverzeichnis

Atmel, AVR Enhanced Risc Microcontroller Data Book, May 1997.

Atmel, Development Tools User Guide.

Atmel, Memory Data Book.

Dallas Semiconductor, Automatic Identification Data Book, 1994/1995

Dallas Semiconductor, Book of DS19xx Touch Memory Standards, 1994

Hitachi LCD Controller/Driver LSI, Data Book, 1995.

Horninger, K.: Integrierte MOS-Schaltungen. Halbleiter-Elektronik Band 14, Springer Verlag, 1987.

ISO 7810-1: Identification cards – Physical characteristica.

ISO 7811-1: Identification cards – Recording technique – Embossing.

ISO 7811-2: Identification cards – Recording technique – Magnetic stripe.

ISO 7811-3: Identification cards – Recording technique – Location of embossed characters.

ISO 7811-4: Identification cards – Recording technique – Location of read only magnetic track 1+2.

ISO 7811-5: Identification cards – Recording technique – Location of read only magnetic track 3.

ISO/IEC 7816-2: Identification cards – Integrated circuit(s) cards with contacts – Part 2: Dimension and location of the contacts.

ISO/IEC 7816-3: Identification cards – Integrated circuit(s) cards with contacts – Part 3: Electronic signals and transmission protocols.

Krings, G.: Intelligente Speicherchips für Chipkarten. Siemens Components 31, Heft 6, 1993, S. 223...227.

Meyer, C., Volpe, S., Volpe, F.P.: Plaste und Elaste, Interna der wichtigsten Chipkarten. c't 12/94, Verlag Heinz Heise Hannover, 1994, S. 310...318.

Volpe, S.: Eingeholt, Neue Konkurrenz bei 8-Bit-Controllern. ELRAD 6/97, Verlag Heinz Heise Hannover, 1997, S. 78...80.

Volpe, F.P.; Volpe, S.: PIC-µC-Praxis, Assembler, Hardwaretools und Anwendungen. Elektor, Aachen, 1996.

Volpe, F.P.; Volpe, S.: Chipkarten, Grundlagen, Technik, Anwendungen. Verlag Heinz Heise Hannover, 1996.

Volpe, F.P.; Volpe, S.: Magnetkarten, Grundlagen, Technik, Anwendungen. Verlag Heinz Heise Hannover, 1995.

Volpe, F.P.; Volpe, S.: Knopfzellen, Grundlagen und PC-Anschluß für Touch-Memories DS199x. ELRAD 10/95, Verlag Heinz Heise Hannover, 1995, S. 63...67.

Volpe, F.P.; Volpe, S.: Telefonmarke, BASIC-Briefmarke II als Telefonkartenleser. ELRAD 12/95, Verlag Heinz Heise Hannover, 1995, S. 40...43.

Stichwortverzeichnis

Symbole

1-Draht-Protokoll 199
3 V-Lithiumzelle 199
3-Level Hardware-Stack 12

A

AA (Authorization Accepted Flag) 206, 207
absolute Adresse 172
Abtasten 203, 204
Abtastfenster 204
ACK (Acknowledge) 217
Acknowledge (ACK) 217
Acknowledge-Polling 221
ACO (Analog Comparator Output) 29
ACSR (Analog Comparator Control and Status Register 30
ADC (Analog Digital Converter) 30
ADC Rd,Rr 46
ADC-Status-Register (ADCSR) 30
ADCSR (ADC-Status-Register) 30
ADD Rd,Rr 47
Add-Only-Memory 201
ADIW Rdl,K 48
Allzweckregister (GPR) 14
Analog Comparator Control and Status Register (ACSR) 30
Analog Comparator Output (ACO) 29
Analog Digital Converter (ADC) 30
Analog-Komparator 29
Analog/Digital-Wandler (ADC) 30
AND Rd,Rr 49
ANDI Rd,K 50
Applikationsschriften 273
ASR Rd 51
Assembler-Sourcen 276
Asynchrone serielle Schnittstelle (UART) 26
AT89C2051 163
AT90S1200 10, 11
AT90S2313 10, 11
AT90S2323 10, 11
AT90S2343 10, 11
AT90S4414 10, 11
AT90S4434 11, 12
AT90S8515 10, 11
AT90S8535 11, 12
ATmega103 11, 12
ATmega603 11, 12
Atmel GmbH 277
Ausdrücke 171
Ausgangstreiber, Touch-Memories 200
Authorization Accepted Flag (AA) 206, 207
AVR AT90ICEPRO Emulator 181
AVR-Assembler 167, 174
AVR-Programmer 181
AVR-Simulator 175
AVR-Studio 167, 175, 176, 177, 192
AVR-Studio Befehle 179
AVR-Studio, Programm-Fenster 178
AVRProg 187, 188

B

Baudratengenerierung 28
BCLR bit 52
Befehlssatz 45
bits per inch (bpi) 249
BLD Rd,bit 53
bpi (bits per inch) 249
BRBC bit,offset 54
BRBS bit,offset 55
BRCC offset 56
BRCS offset 57
Breakpoints (Haltepunkte) 192
BREQ offset 58
BRGE offset 59
BRHC offset 60
BRHS offset 61
BRID offset 62
BRIE offset 63
BRLO offset 64
BRLT offset 65
BRMI offset 66

BRNE offset 67
BRPL offset 68
BRSH offset 69
BRTC offset 70
BRTS offset 71
BRVC offset 72
BRVS offset 73
BSET bit 74
BST Rd,bit 75
Busy-Flag 231
Byte-Read 220
Byte-Write 219

C

CALL addr 76
Capture-Funktion 36
Card Load Signal (CLS) 254
CBI Port,bit 77
CBR Rd,mask 78
CG-RAM 231
CG-ROM (Caracter Generator ROM) 231
Character Generator ROM (CG-ROM) 231
Chipkarten 240
Chipkarten Programmierspannung 240
Chipmodul 240
CLC 79
CLH 80
CLI 81
CLN 82
CLR Rd 83
CLS 84
CLS (Card Load Signal) 254
CLT 85
CLV 86
CLZ 87
COM Rd 88
Compare-Funktion 36
CP Rd,Rr 89
CPC Rd,Rr 90
CPI Rd,K 91
CPSE Rd,Rr 92
CRC (Cyclic Redundancy Check)-Prüfsumme 268
CRC-8 (Cyclic Redundancy Check) 200
CRC-Akkumulator 269
Cyclic Redundancy Check (CRC-8) 200
Cyclic Redundancy Check-Prüfsumme (CRC) 268

D

Dallas Semiconductor 199, 277
Data Direction Register (DDR) 23
Data Display RAM (DD) 230
Data Over Voice-Modem (DOV) 243
Datenblätter 275
Datenspeicher 16
Datentypen 174
Datentypen-Syntax 174
Datenübertragungsrate I²C-Bus 222
DB-Direktive 174
DD (Data Display RAM) 230
DDR (Data Direction Register) 23
DEC Rd 93
Device Select Code 215
direkte Werte 173
Direktiven 169
DOV (Data Over Voice)-Modem 243
DS 198x 201
DS 1990A 199, 201
DS 1991 199, 201
DS 1992 201
DS 1993 201
DS 1994 201
DS 1995 201
DS 1996 199, 201

E

E/S-Register 205
Echtzeit (Real-Time) 192
Echtzeituhr (RTC) 32
EEPROM 17
EEPROM-Datenspeicher 12
EEPROM-File 167
Einzelschritt (Single-Step) 192
Elektronik Laden Detmold 195, 278
ELPM 94
Emulationsmodus 192
Emulator AVR AT90ICEPRO 192
End-Offset 205
EOR Rd,Rr 95
Equinox Technologies 278
Experimentierplatine 206
externes Taksignal 44

F

F2F (Frequency/Double Frequency) 254
F2F-Decoder 254
F2F-Modulation 251, 254
Family Code 200
Fehlererkennung 268
Flash-Speicher 14
Flußdiagramm 207
Frequency/Double Frequency (F2F) 254
Funktionen 175
Future Electronics Deutschland GmbH 278

G

General Purpose Register (GRP) 14, 18
Generatorpolynom 268
GPR (General Purpose Register) 14, 18

H

Haltepunkte (Breakpoints) 192
Hardware-Stack 10, 39
Harvard-Architektur 12
HD44780 (LCD-Controller) 229
Herstellercodierung 241
Hewlett-Packard GmbH 279

I

I/O-Leitungen 23
I/O-Ports 23
I/O-Register 20
I²C-Bus 215
IAR Systems GmbH 279
ICALL 96
IJMP 97
IN Rd,Port 98
In-Circuit 15, 181
In-Circuit-Emulator 178
INC Rd 99
Include-Dateien 169, 274
INELTEK GmbH 279
Ingenierbüro Yahya 280
Interrupt-Vektor 42
ISO 7810 248
ISO 7811-2 250
ISO 7811-4 249
ISO 7811-5 249
ISO 7816-3 241

J

JMP addr 100

K

Kommentare 170

L

Ladungspumpe 240
LCD-Anzeige 229
LCD-Controller 229
LD Rd,X 101
LD Rd,X+ 102
LD Rd,-X 103
LD Rd,Y 104
LD Rd,Y+ 105
LD Rd,-Y 106
LD Rd,Z 107
LD Rd,Z+ 108
LD Rd,-Z 109
LDD Rd,Y+ offset 110
LDD Rd,Z+ offset 111
LDI Rd,K 112
LDS Rd,addr 113
Lesekopf 254
Listing-File 167
Logikanalysator HP54645D 194
Longitudinal Redundancy Check (LRC) 249
LPM 114
LRC (Longitudinal Redundancy Check) 249
LSL Rd 115
LSR Rd 116

M

Magnetkarte 248
Magnetkartenleser 248
Magnetstreifen 248
Marke 172
Master 201
Master In Slave Out (MISO) 24
Master Out Slave In (MOSI) 24
MAX232 183
MicroLAN 200
MISO (Master In Slave Out) 24
MOSI (Master Out Slave In) 24
Most Significant Bit (MSB) 217
MOV Rd,Rr 117
MSB (Most Significant Bit) 217
Multi-Master-Bus 216
Multibyte-Write 220
Multibyte/Page-Write 219

N

NEG Rd 118
Netzteil 186

NOP ... 119
Nulldruckfassung ... 185

O

Objekt-File ... 167
odd (ungerade Parität) ... 251
OF (Overflow Flag) ... 206
Open-Collector ... 216
Open-Drain ... 200, 216
OR Rd,Rr ... 120
ORI Rd,K ... 121
OUT Rd,Port ... 122
Overflow Flag (OF) ... 206

P

Page ... 219
Page-Write ... 220
Parasitic Power ... 199
Partial Byte Flag (PF) ... 206
PC (Program Counter) ... 12, 38
Pegelwandlung ... 183, 262
Peripherie-Bausteine ... 215
Personalisierdaten ... 241
PF (Partial Byte Flag) ... 206
Philips ... 215
PLCC-Gehäuse ... 185
Polysilizium-Bahnen ... 200
POP Rd ... 123
POR (Power-On-Reset) ... 41
Power-On-Reset (POR) ... 41
Presence Pulse ... 202
Program Counter (PC) ... 12, 38
Programmsammlung ... 195
Programmspeicher ... 12, 14
Programmzähler (PC) ... 12, 38
Projekte ... 195
PUSH Rd ... 124
PWM-Modus ... 38

Q

Quarz-Oszillator ... 43

R

R/W (Read/Write) ... 218
RC Enable (RCEN) ... 43
RC-Oszillator, interner ... 43
RCALL offset ... 125
RCEN (RC Enable) ... 43
RCP (Read Clock Pulse) ... 254
RDP (Read Data Pulse) ... 254
Read Clock Pulse (RCP) ... 254
Read Data Pulse (RDP) ... 254
Read-Data Time Slot ... 204
Read/Write (R/W) ... 218
Real Time Clock (RTC) ... 32
Real-Time (Echtzeit) ... 192
Receiver (Empfänger) ... 215
Reduced Instruction Set Computer (RISC) ... 12
Reset-Vektor ... 42
RET ... 126
RETI ... 127
RISC (Reduced Instruction Set Computer) ... 12
RJMP offset ... 128
ROL Rd ... 129
ROR Rd ... 130
RS-232 ... 183, 262
RTC (Real Time Clock) ... 32
Rücksetzsequenz ... 231

S

SBC Rd,Rr ... 131
SBCI Rd,K ... 132
SBI Port,bit ... 133
SBIC Port,bit ... 134
SBIS Port,bit ... 135
SBIW Rdl,K ... 136
SBR Rd,mask ... 137
SBRC Rr,bit ... 138
SBRS Rr,bit ... 139
Schreibdichte ... 250
Schreibschutz-Eingang ... 218
SCK (Serial Clock) ... 24
SCL (Serial Clock) ... 215
SDA (Serial Data) ... 215
SEC ... 140
SEH ... 141
SEI ... 142
selbstgetaktete Daten ... 254
SEN ... 143
Sender (Transmitter) ... 215
Sequential-Read ... 220
SER Rd ... 144
Serial Clock (SCK) ... 24
Serial Clock (SCL) ... 215
Serial Data (SDA) ... 215
Serial Peripheral Interface (SPI) ... 24
serielle EEPROMs ... 215
serielle Schnittstelle ... 183, 261
Seriennummer ... 200

SES ... 145
SET ... 146
SEV ... 147
SEZ ... 148
Simulationsmodus ... 192
Single-Step (Einzelschritt) ... 192
Slave ... 201
Slave Select (SS) ... 24
SLEEP ... 149
Softwaretools ... 274
SP (Stack-Pointer) ... 12
SPI (Serial Peripheral Interface) ... 24
SPI-Schnittstelle ... 24
SRAM-Datenspeicher ... 12
SREG (Status-Register) ... 22
SS (Slave Select) ... 24
ST X,Rr ... 150
ST X+,Rr ... 151
ST -X,Rr ... 152
ST Y,Rr ... 153
ST Y+,Rr ... 154
ST -Y,Rr ... 155
ST Z,Rr ... 156
ST Z+,Rr ... 157
ST -Z,Rr ... 158
Stack ... 38, 40
Stack-Pointer (SP) ... 12, 39, 40
Startsequenz ... 216
Status-Register (SREG) ... 22
STD Y+ offset,Rr ... 159
STD Z+ offset,Rr ... 160
Stopsequenz ... 216
STS addr,Rr ... 161
SUB Rd,Rr ... 162
SUBI Rd,K ... 163
SWAP Rd ... 164
Symbole ... 172

T

Target-Adresse ... 205
TCCRx (Timer Counter Control Register) ... 36
Telefonkarte ... 240
Telefonkartenleser ... 240
Time-Slots ... 201
Timer Counter Control Register (TCCRx) ... 36
Timer/Counter ... 32
Toleranzbereich ... 203
Touch-Memories ... 199
Trägermaterial ... 248
Transmitter (Sender) ... 215
TST Rd ... 165

U

UART (Universal Asynchronous Receiver Transmitter) ... 261
ungerade Parität (odd) ... 251
Universal Asynchronous Receiver Transmitter (UART) ... 261
unmittelbare Werte ... 173
Upgrade-Kits ... 193

V

von Neumann-Architektur ... 12

W

Watchdog-Timer (WDT) ... 38
Watchdog-Timer-Control-Register (WDTCR) ... 38
WDR ... 166
WDT (Watchdog-Timer) ... 38
WDTCR (Watchdog-Timer-Control-Register) ... 38
Wechseltaktschrift ... 254
Wired-AND ... 200
Wired-AND-Bus ... 216
WP (Write-Protect) ... 218
Write Scratchpad-Befehl ... 205
Write-One Time Slot ... 204
Write-Protect (WP) ... 218
Write-Zero Time Slot ... 204

Z

Zeichensatzgenerator ... 231